Advanced Algorithms II

Summer 2018 • Lecture 08/14

Final

Final

Bishop Auditorium 8:30 to 11:30 a.m. this Friday 8/17.

You can use four 1 sided-sheets of paper.

Final Review

Next class, I'll review what you need to know!

Outline for Today

Advanced graph algorithms

- Karger's Algorithm for finding global minimum cuts
- Ford-Fulkerson for finding s-t minimum cuts

Karger's Algorithm

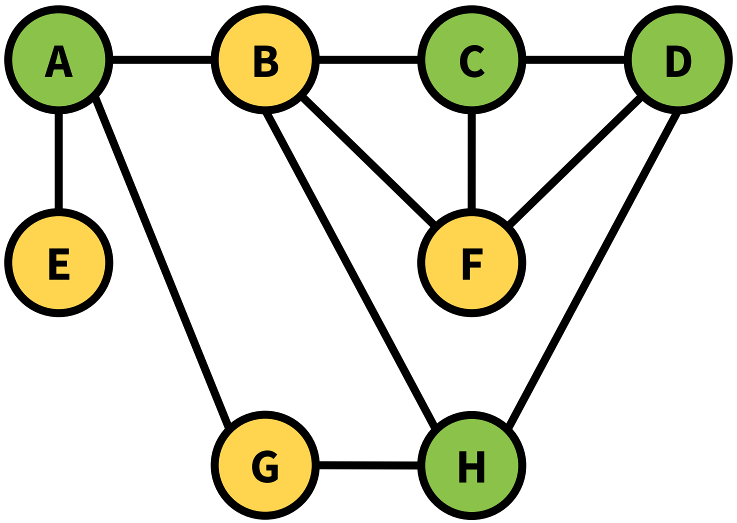
Cuts

A **cut** is a partition of the vertices into two nonempty parts.

Cuts

A **cut** is a partition of the vertices into two nonempty parts.

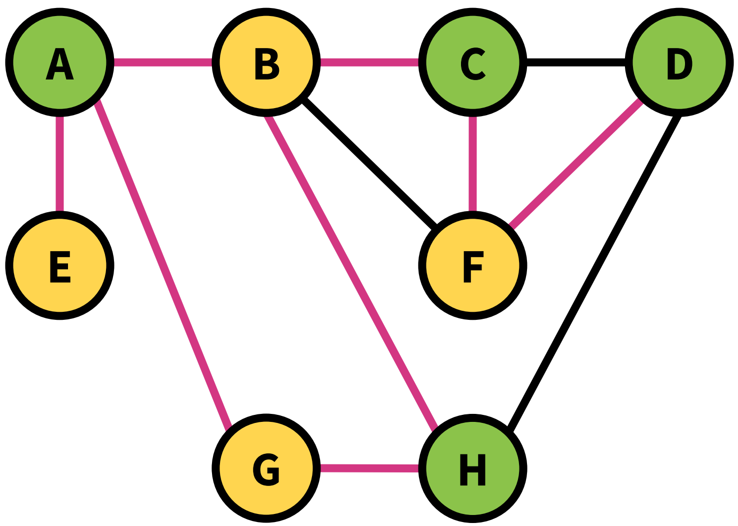
e.g. This is the cut "{A, C, D, H} and {B, E, F, G}".



Cuts

A **cut** is a partition of the vertices into two nonempty parts.

e.g. This is the cut "{A, C, D, H} and {B, E, F, G}".

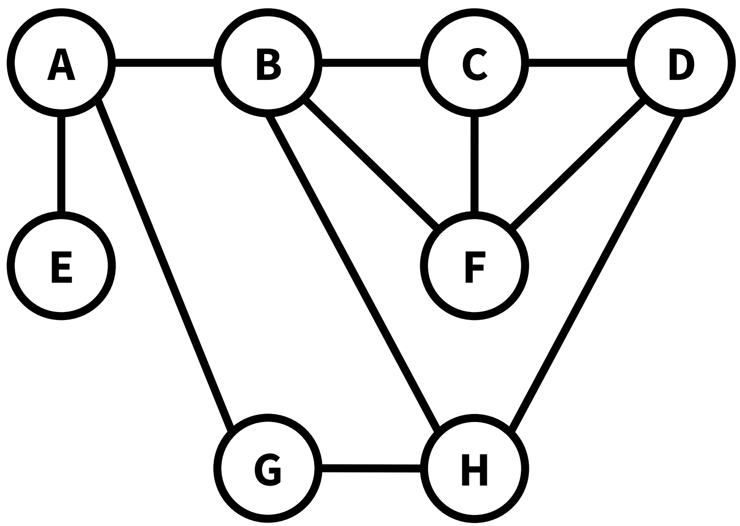


Edges that **cross the cut** go from one part to the other.

e.g. These edges cross the cut.

Cuts

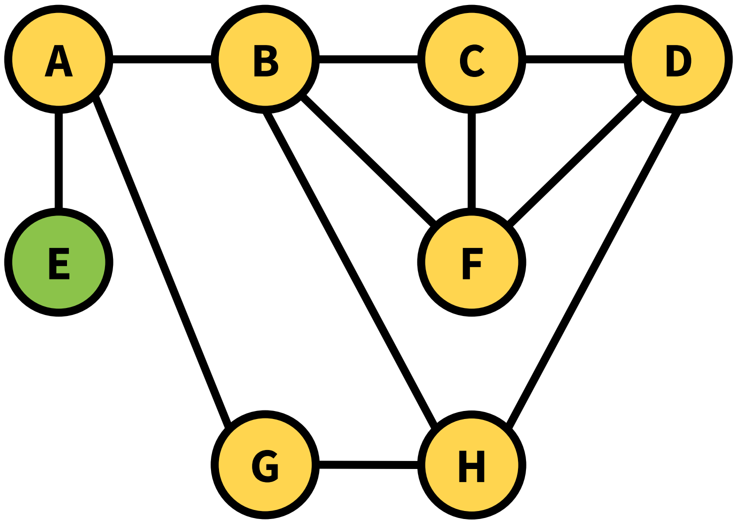
A **global minimum cut** is a cut that has the fewest edges possible crossing it.



Cuts

A **global minimum cut** is a cut that has the fewest edges possible crossing it.

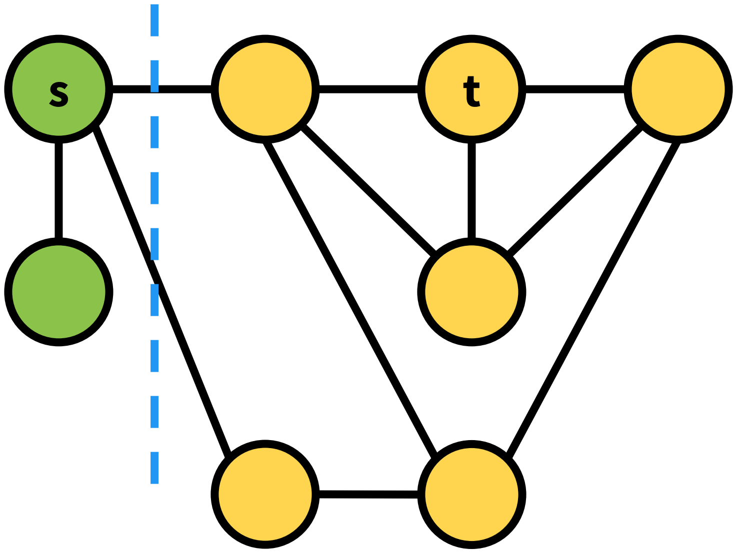
e.g. The global minimum cut is “{A, B, C, D, F, G, H} and {E}”.



Cuts

We'll talk about **minimum s-t cuts**, which separate specific vertices **s** and **t**.

e.g. The s-t minimum cut is this cut.



Global Minimum Cuts

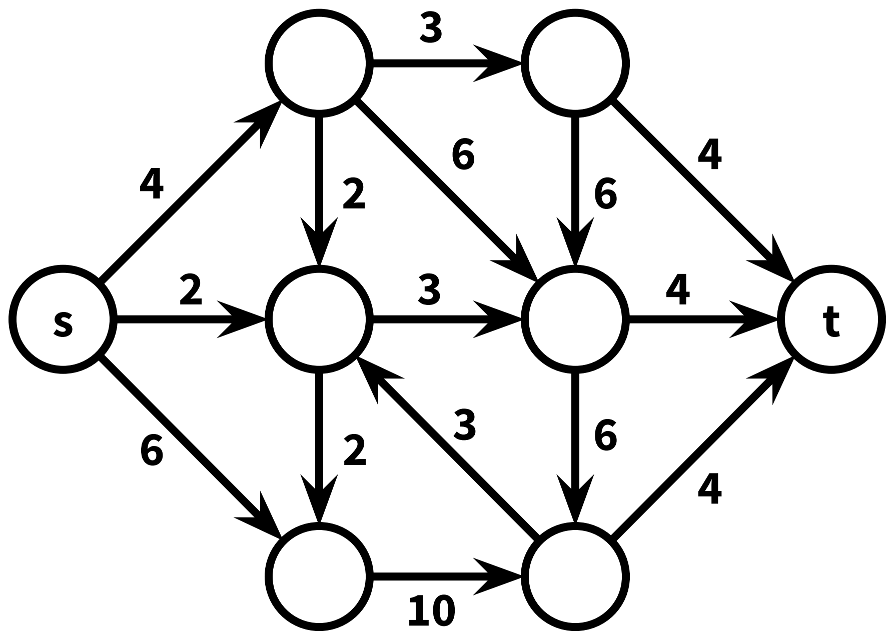
Why might we care about global minimum cuts?

Application: Image segmentation

Minimum Cuts

Graphs are directed and edges have “capacities” (weights).

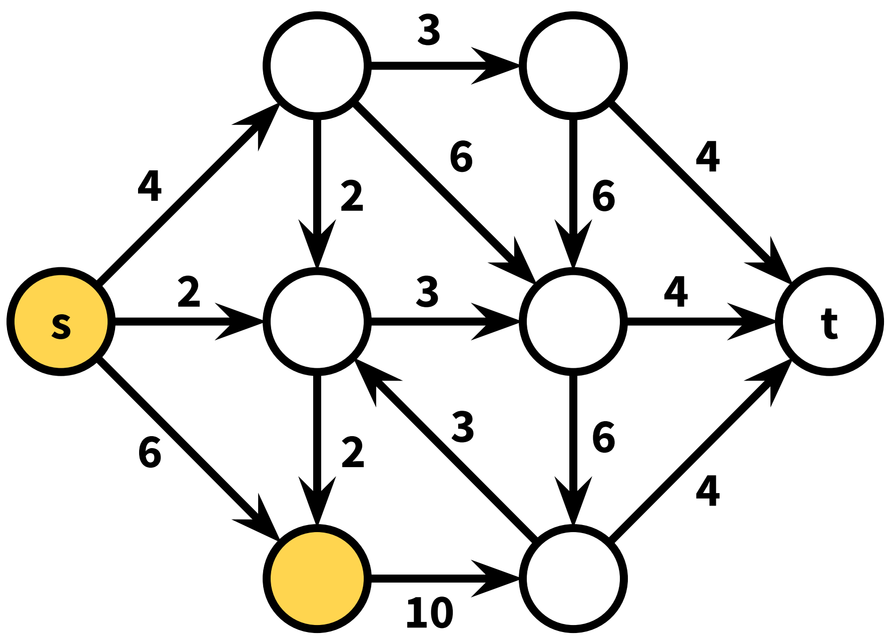
- There’s a special “source” vertex s with only outgoing edges and a “sink” vertex t with only incoming edges.



Minimum Cuts

Graphs are directed and edges have “capacities” (weights).

There’s a special “source” vertex s with only outgoing edges and a “sink” vertex t with only incoming edges.

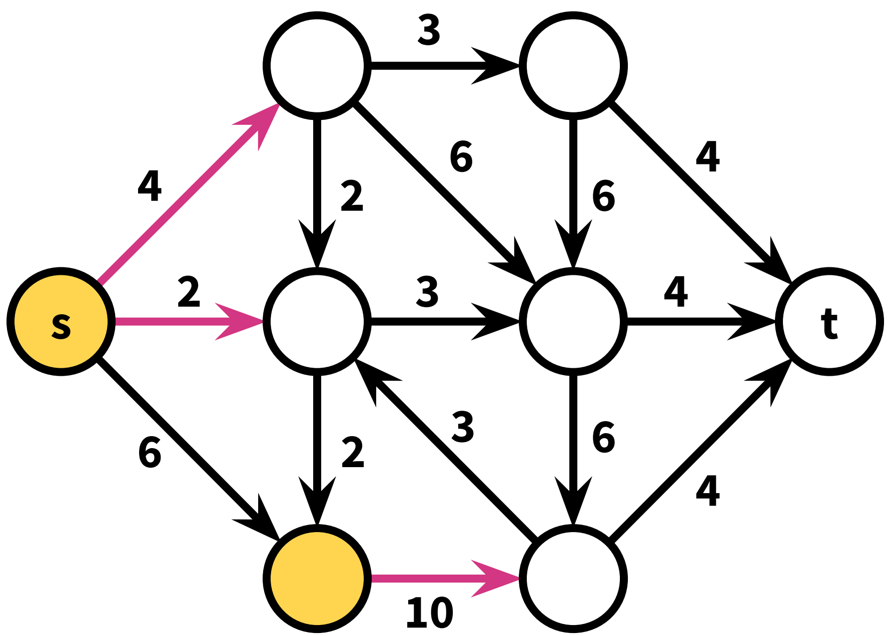


An s-t cut is a cut that separates s from t.

Minimum Cuts

Graphs are directed and edges have “capacities” (weights).

There’s a special “source” vertex s with only outgoing edges and a “sink” vertex t with only incoming edges.



An s-t cut is a cut that separates s from t.

An edge crosses the cut if it goes from s’s side to t’s side.

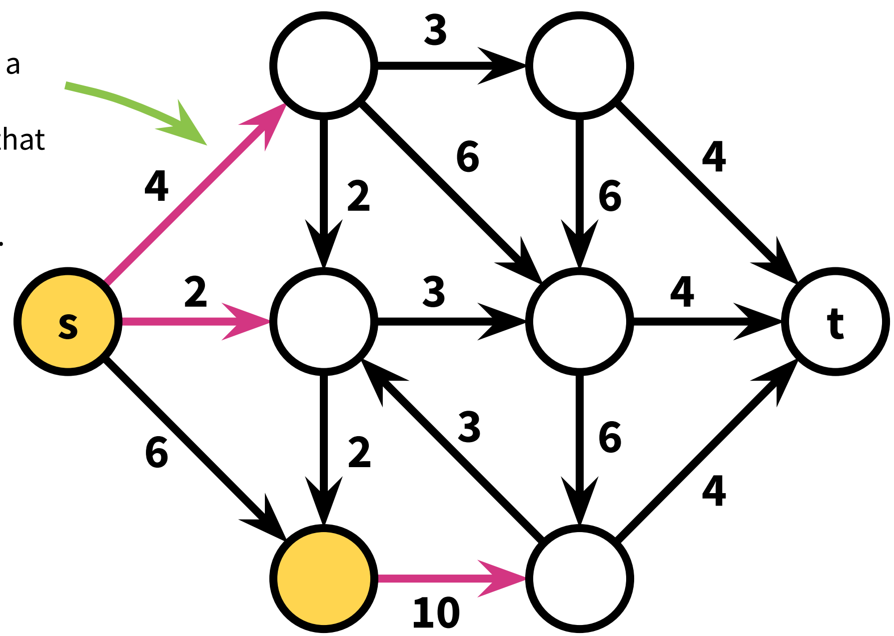
Minimum Cuts

Graphs are directed and edges have "capacities" (weights).

There's a special "source" vertex s with only outgoing edges and a "sink" vertex t with only incoming edges.

The cost (or capacity) of a cut is the sum of the capacities of the edges that cross the cut.

The cost of this cut is 16.

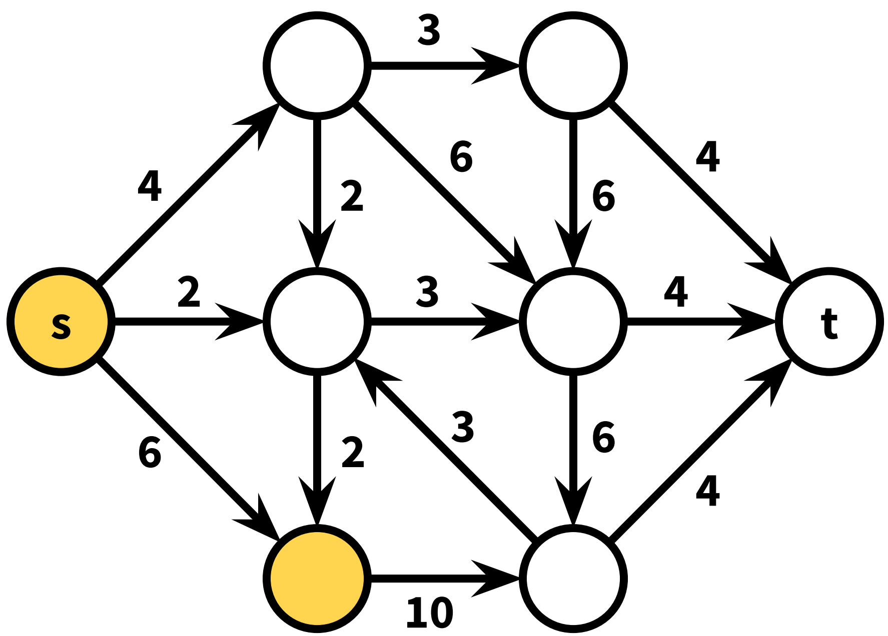


An s-t cut is a cut that separates s from t.

An edge crosses the cut if it goes from s's side to t's side.

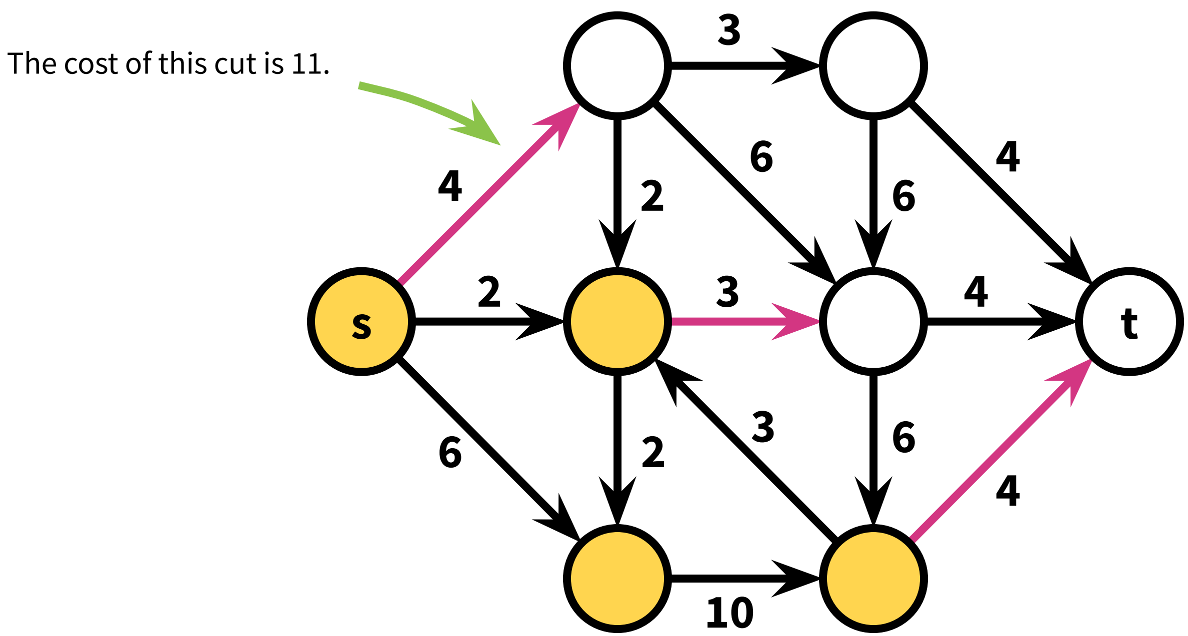
Minimum Cuts

A minimum s-t cut is a cut which separates s from t with minimum capacity.



Minimum Cuts

A minimum s-t cut is a cut which separates s from t with minimum capacity.



Karger's Algorithm

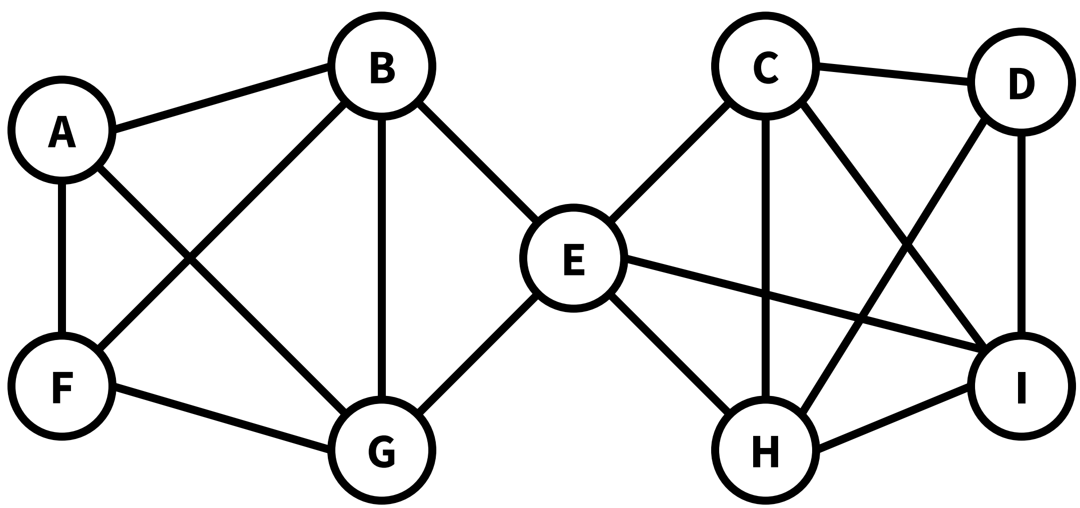
Karger's Algorithm finds global minimum cuts.

- It's a Monte Carlo randomized algorithm! Unlike `quicksort`, which is always correct but sometimes slow, Karger's algorithm is always fast but sometimes Incorrect.
  - For all inputs A, `quicksort` returns a sorted list. For all inputs A, with high probability over the choice of pivots, `quicksort` runs fast.
  - For all inputs G, `karger` runs fast. For all inputs G, with high probability over the randomness in the algorithm, `karger` returns a minimum cut.

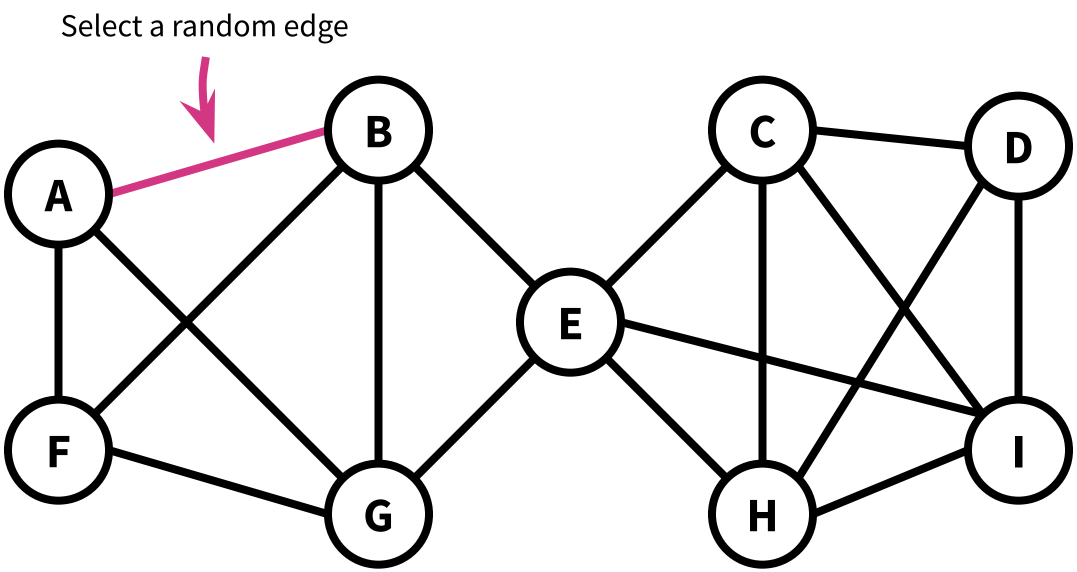
Karger’s Algorithm

The general idea is to pick random edges to “contract” until there are a minimal number of vertices and edges left.

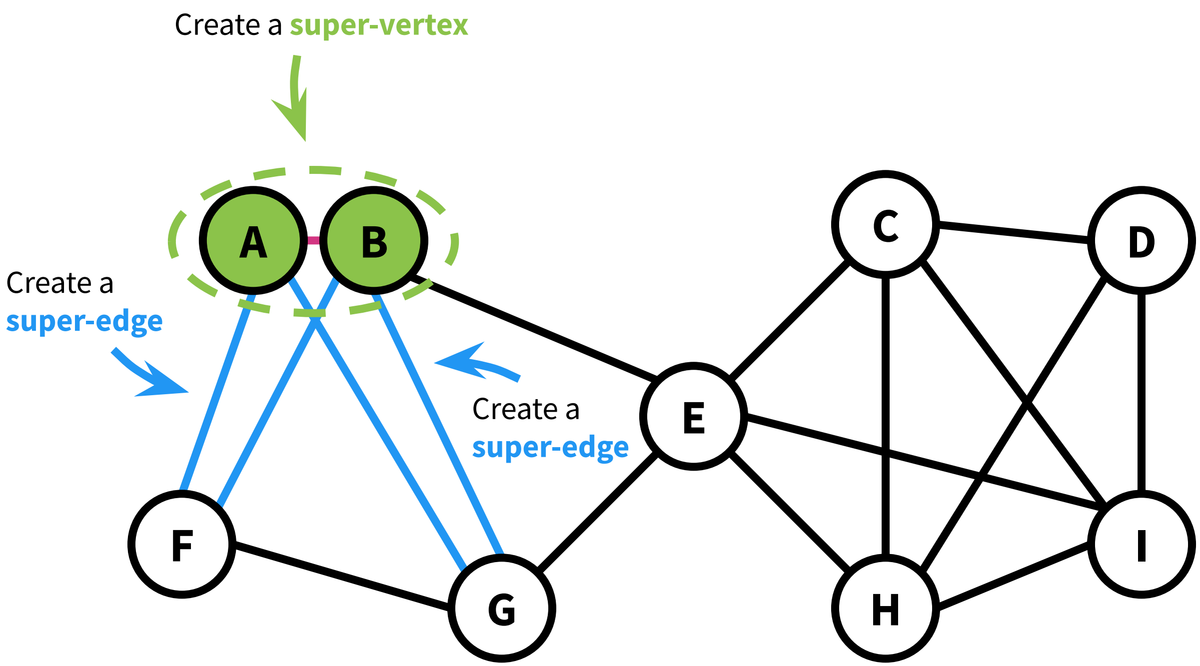
Karger's Algorithm



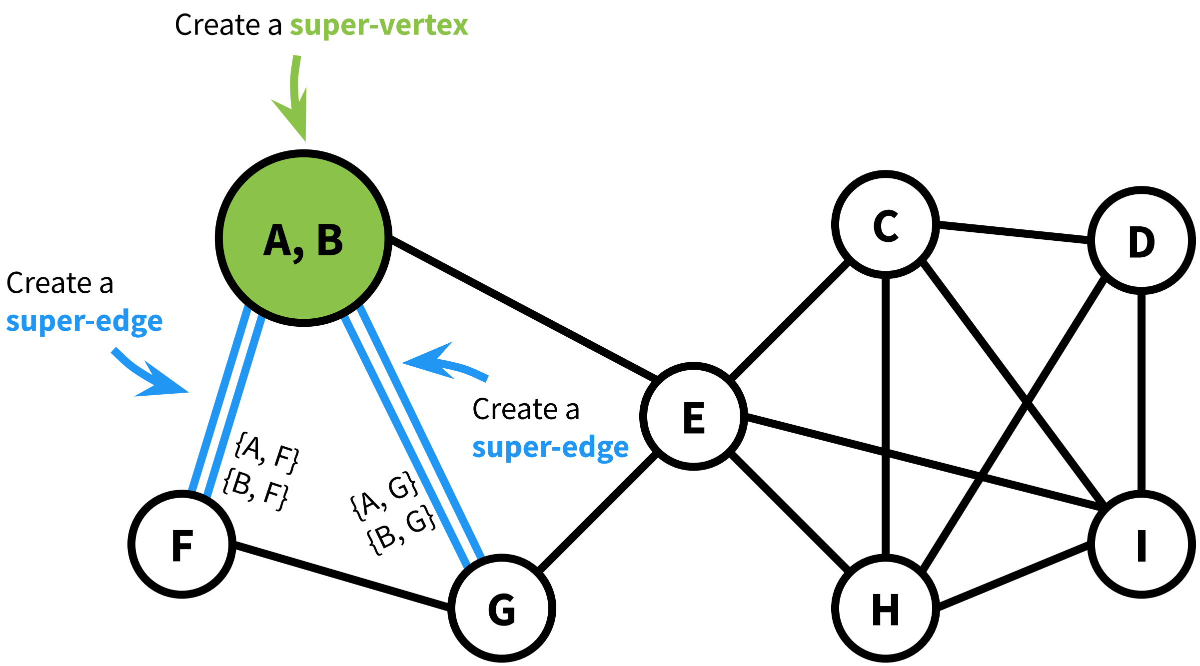
Karger's Algorithm



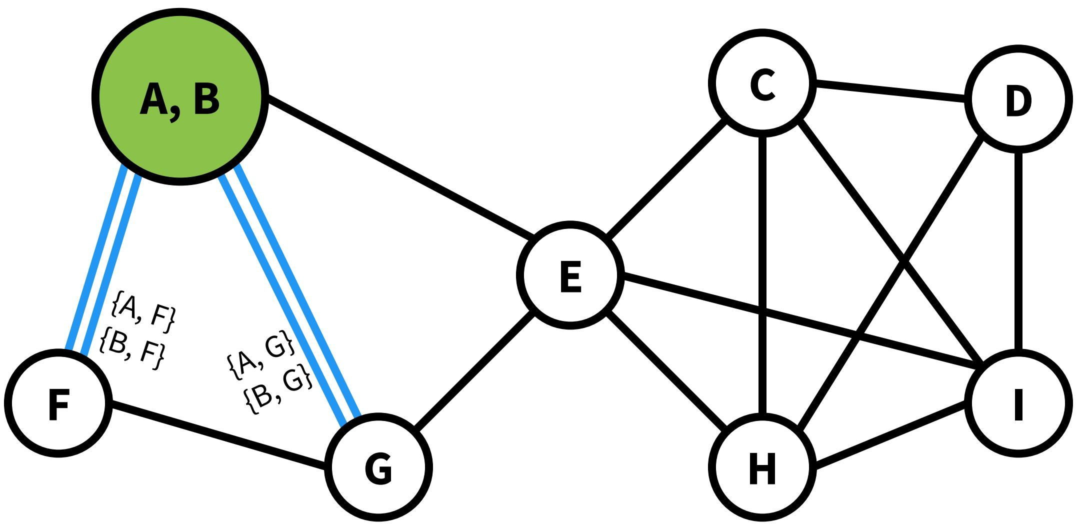
Karger's Algorithm



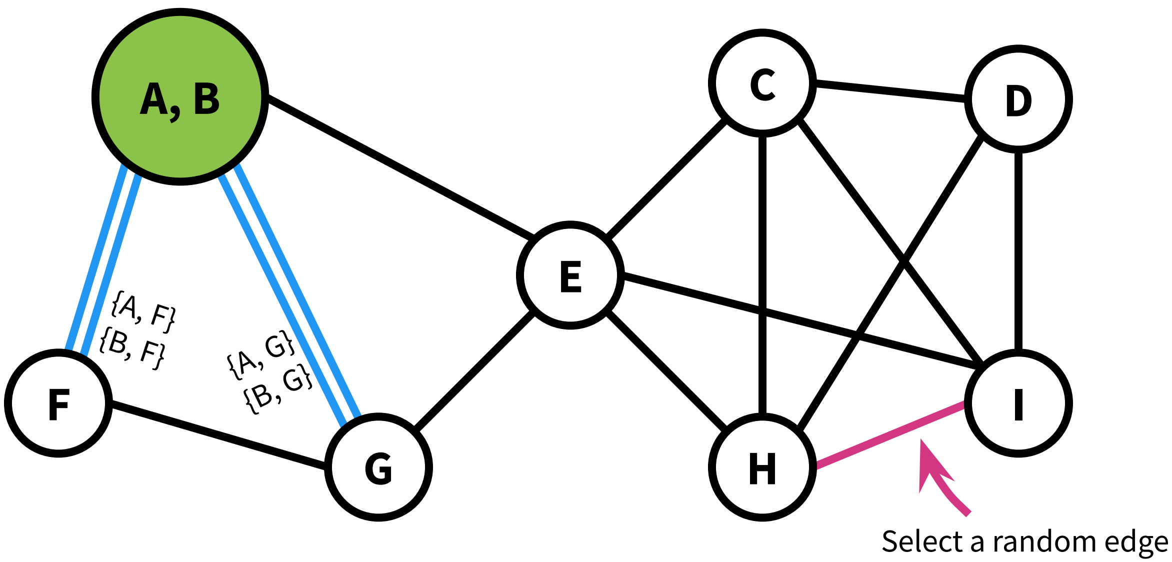
Karger's Algorithm



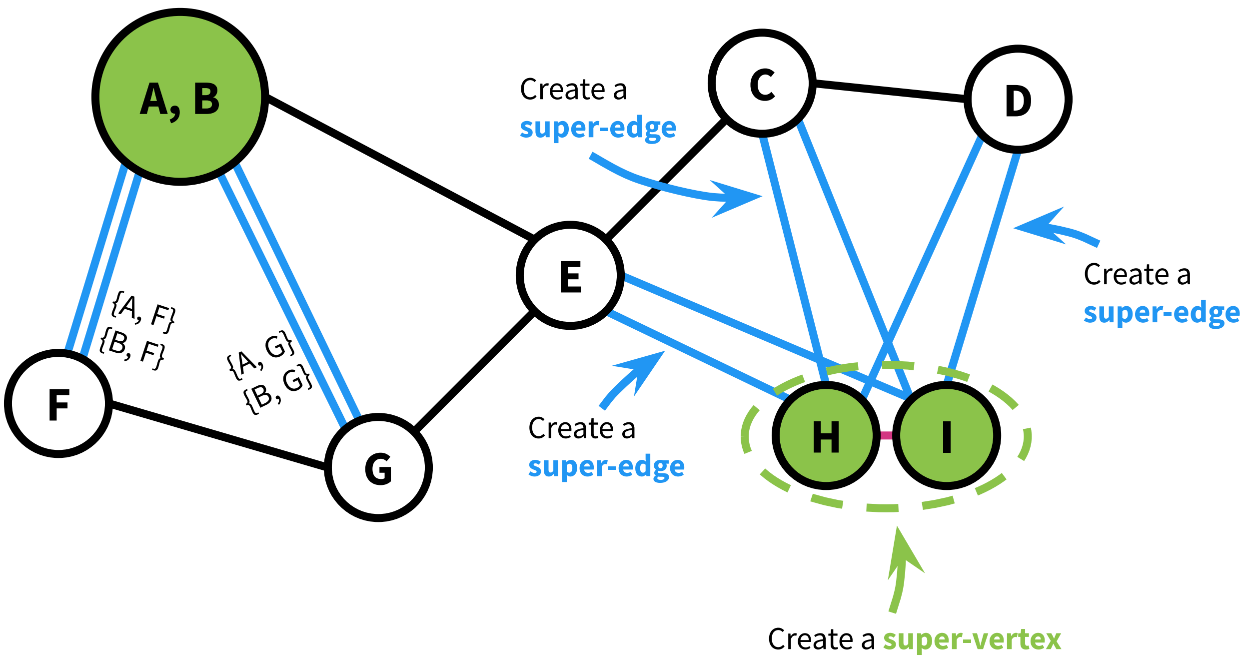
Karger's Algorithm



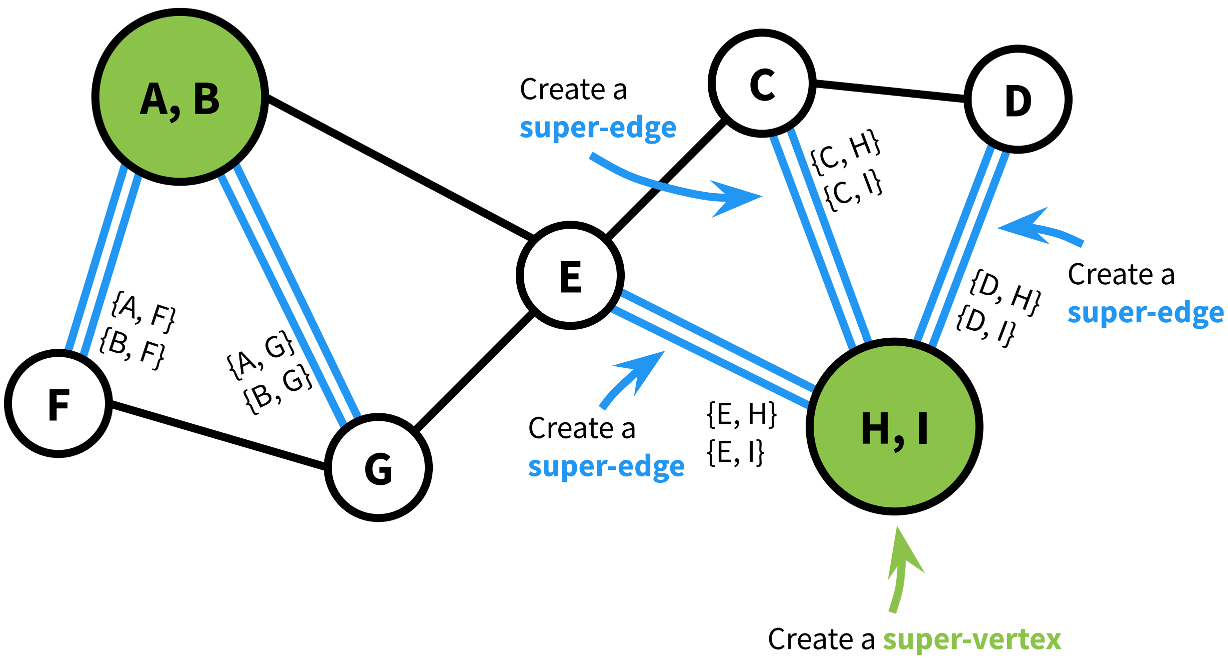
Karger’s Algorithm



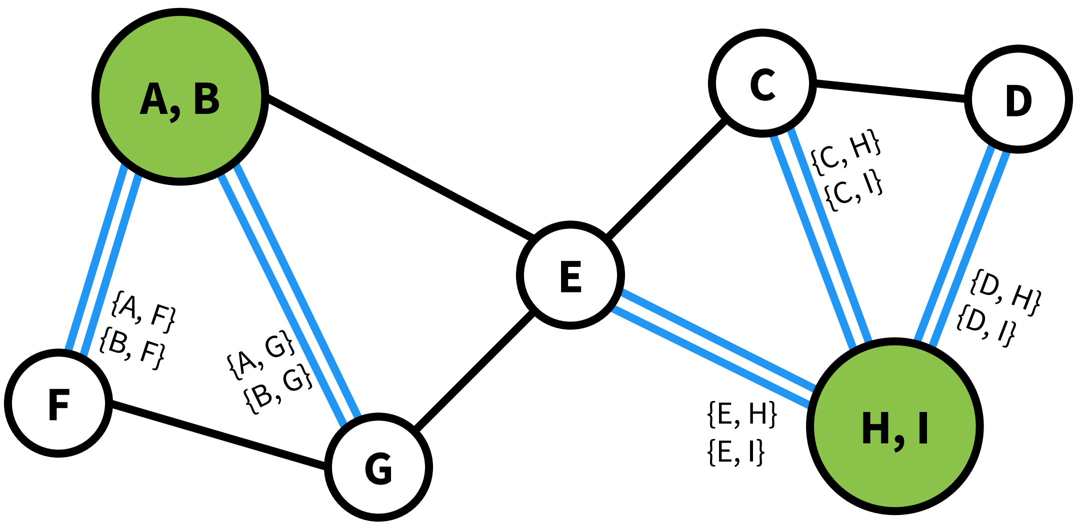
Karger's Algorithm



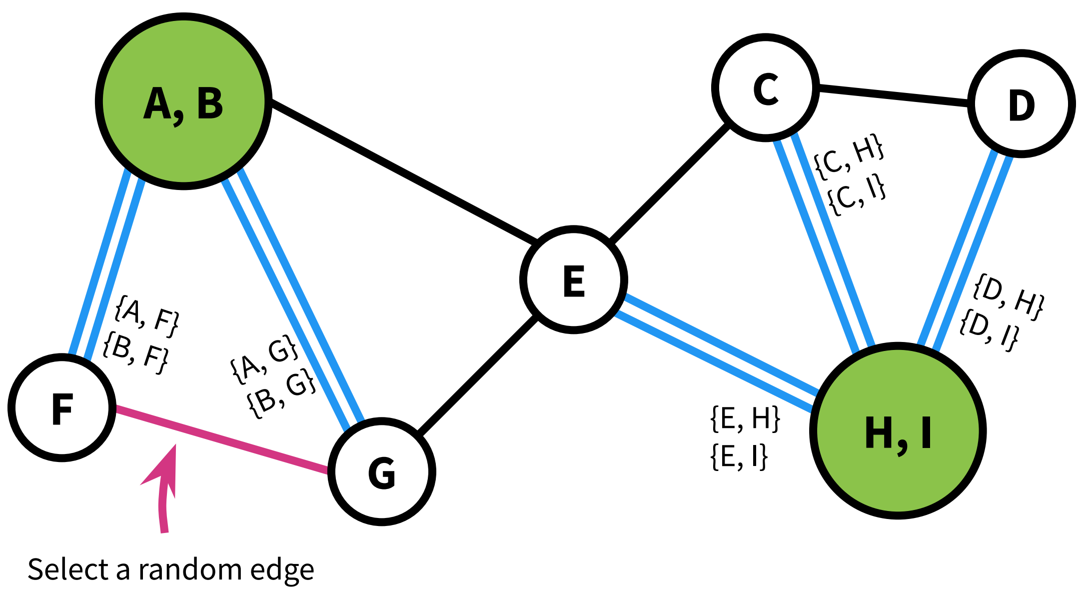
Karger's Algorithm



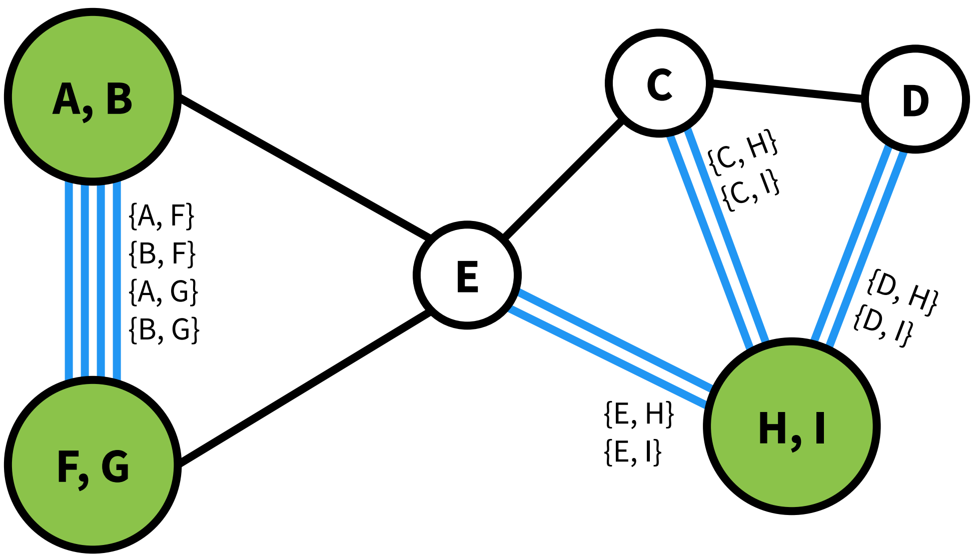
Karger's Algorithm



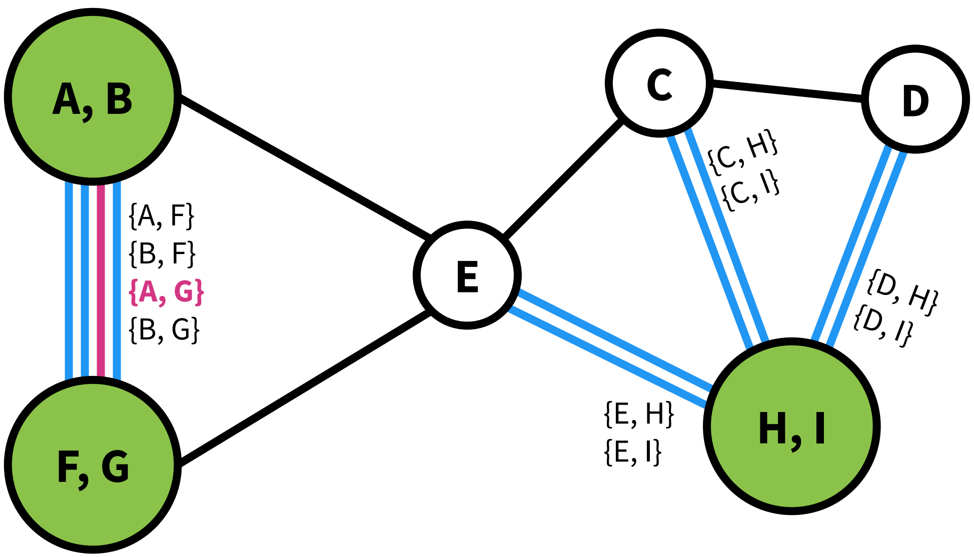
Karger's Algorithm



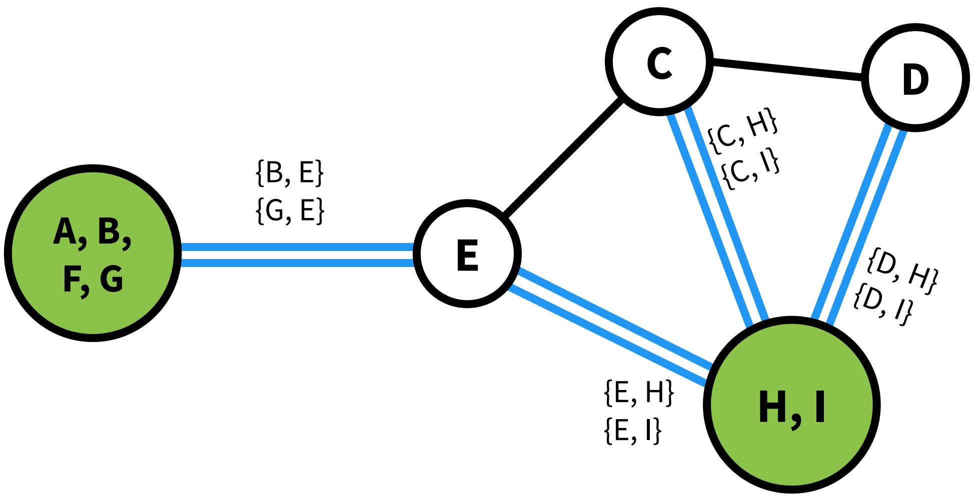
Karger's Algorithm



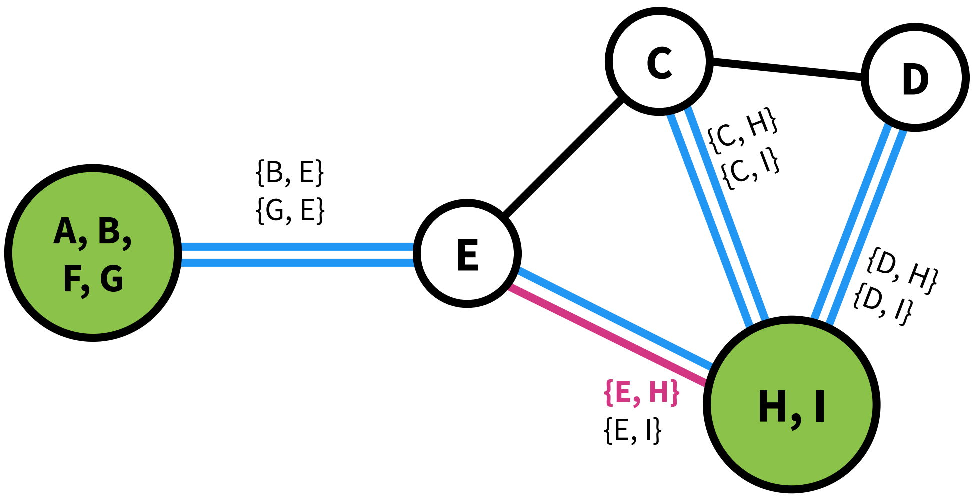
Karger's Algorithm



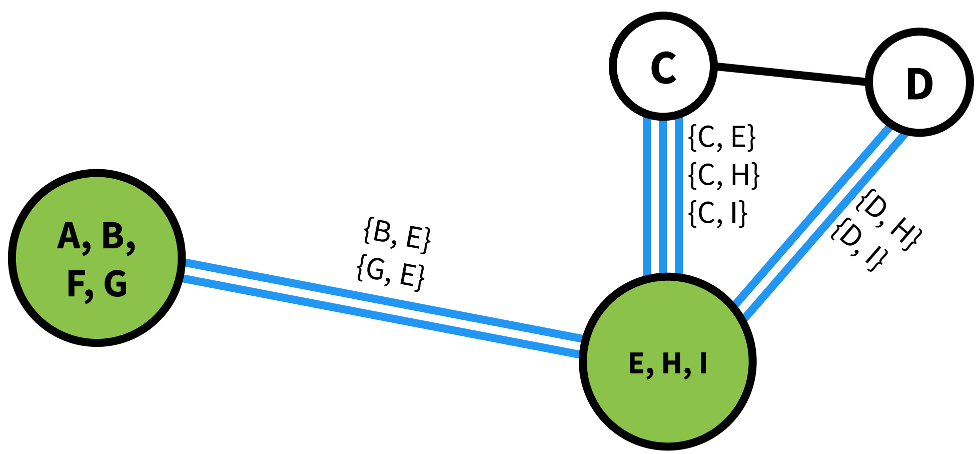
Karger's Algorithm



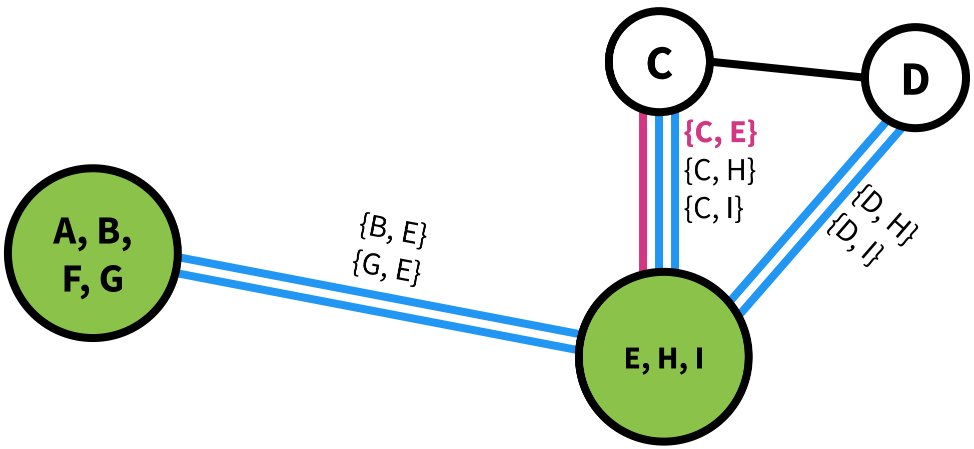
Karger's Algorithm



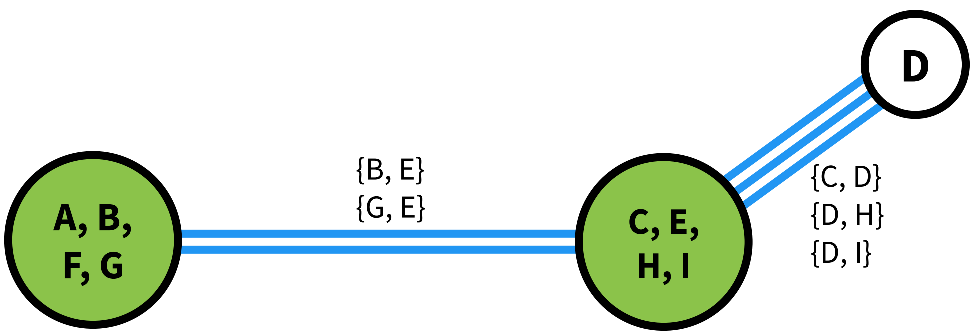
Karger's Algorithm



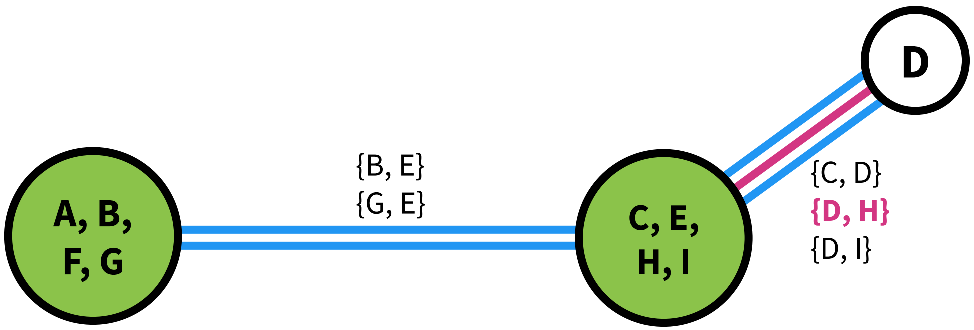
Karger's Algorithm



Karger's Algorithm



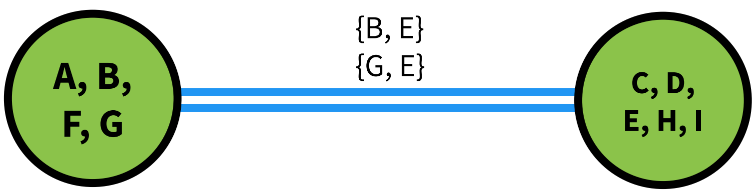
Karger's Algorithm



Karger's Algorithm

The minimum cut is given by the remaining super-vertices.

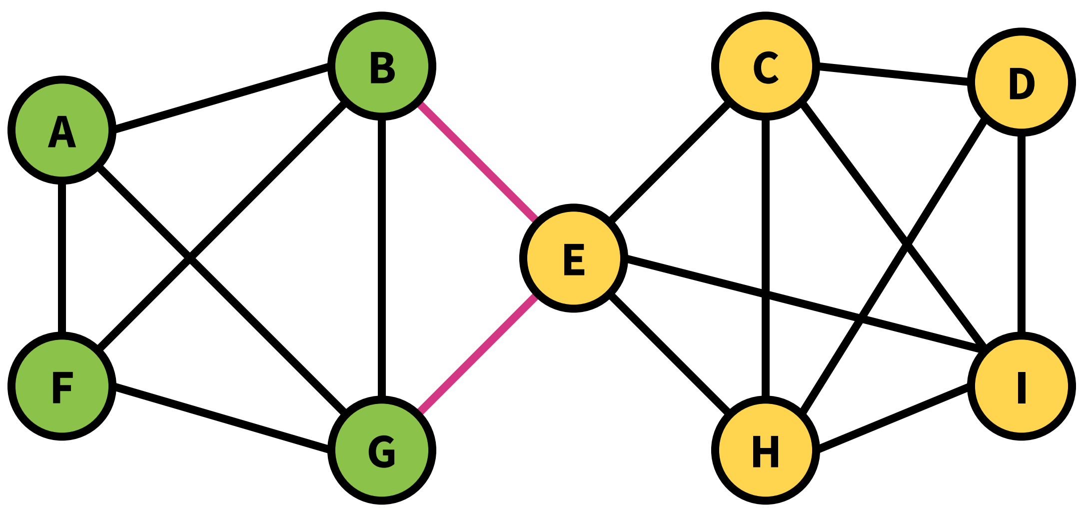
e.g. The cut is "{A, B, F, G} and {C, D, E, H, I}"; the edges that cross this cut are {B, E} and {G, E}.



Karger's Algorithm

The minimum cut is given by the remaining super-vertices.

e.g. The cut is "{A, B, F, G} and {C, D, E, H, I}"; the edges that cross this cut are {B, E} and {G, E}.



Karger's Algorithm

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

Removes all edges in the super-edge between super-vertices u' and v'.

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

Runtime: $O(|V|^2)$

Karger's Algorithm

These are the super-edges.

The while loop executes |V| - 2 times.

Removes all edges in the super-edge between super-vertices u' and v'.

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_u'v' = {(u,v)} for (u,v) E
  E_u'v' = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_u'v'
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_x'w' = E_u'w' ∪ E_v'w'
    Remove u' and v' from G' and add x'
```

This takes O(|V|).

u' is the super-vertex containing u; v' is the super-vertex containing v.

Runtime: $O(|V|^2)$

Karger's Algorithm

```
def karger(G=(V,E)):
  G' = {supervertex(v) for v in V}
  E_{u'v'} = {(u,v)} for (u,v) E
  E_{u'v'} = {} for (u,v) not in E
  F = {{(u,v)} for (u,v) in E}
  while |G'| >= 2:
    {(u,v)} = uniform random edge in F
    merge_supervertices(u, v)
    F = F \ E_{u'v'}
  return cut of the remaining super-vertices

def merge_supervertices(u, v):
  x' = supervertex(u' ∪ v')
  for w' in G' \ {u',v'}:
    E_{x'w'} = E_{u'w'} ∪ E_{v'w'}
    Remove u' and v' from G' and add x'
```

These are the super-edges.

The while loop executes |V| - 2 times.

This takes O(|V|).

Removes all edges in the super-edge between super-vertices u' and v'.

u' is the super-vertex containing u; v' is the super-vertex containing v.

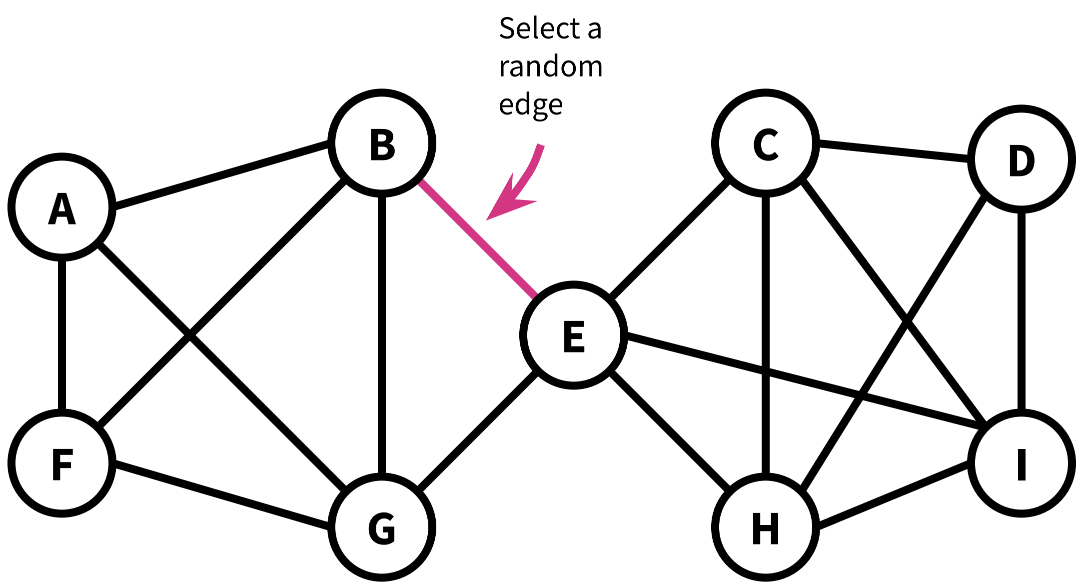
Runtime: $O(|V|^2)$

We can do better with fancy data structures, but this is fine for now.

Karger's Algorithm

We got really lucky!

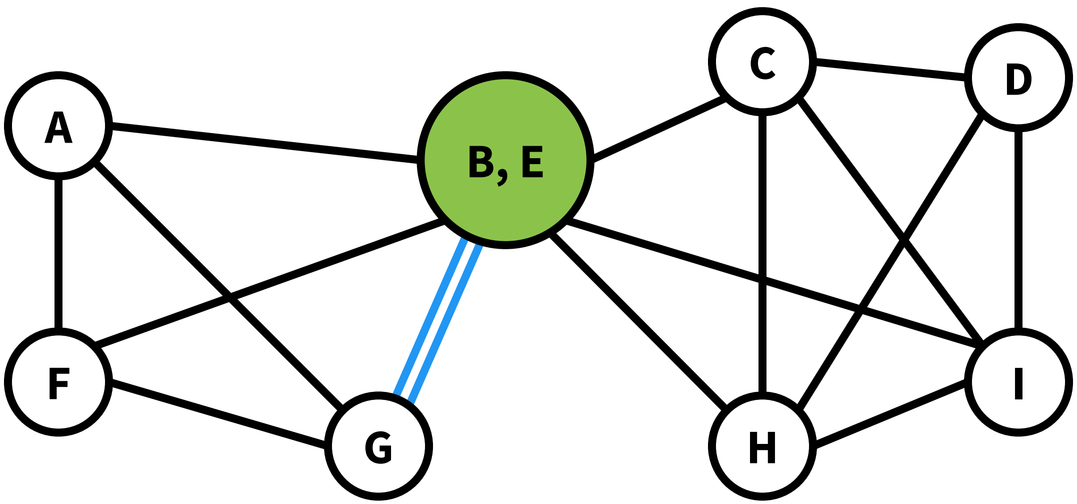
e.g. Suppose we had chosen this edge.



Karger's Algorithm

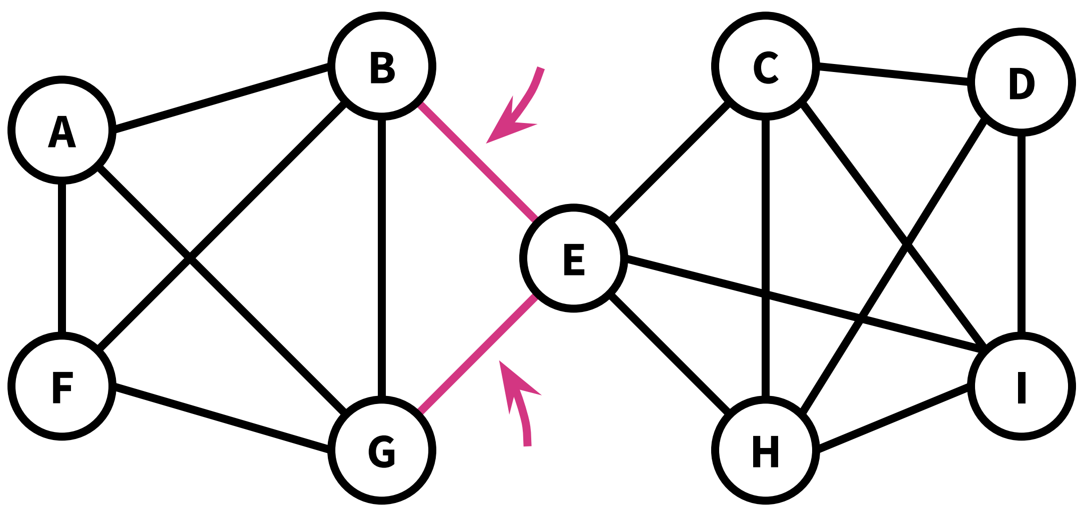
We got really lucky!

- e.g. Suppose we had chosen this edge. Now there's no way to return a cut that separates B and E.



Karger's Algorithm

If fact, if Karger's algorithm ever randomly selects **edges in the min-cut**, then it will be incorrect.



Karger's Algorithm

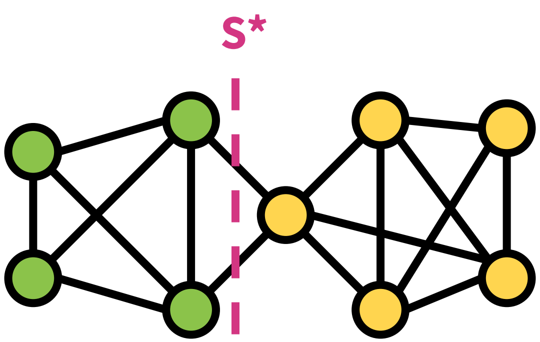
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof:

Suppose S* is a min-cut and suppose we select edges e_1, e_2, …, e_{n-2}.

Then P(karger returns S*) = P(no e_i crosses S*)



Karger's Algorithm

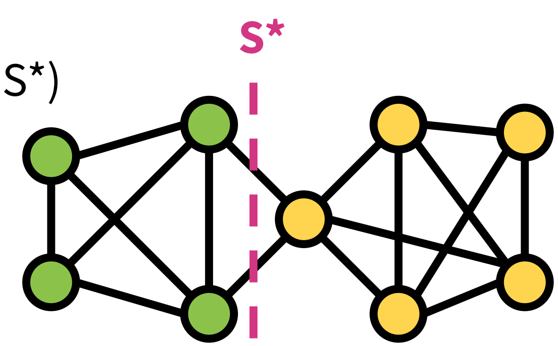
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

Then P(karger returns S*) = P(no e_i crosses S*)
= P(e_1 doesn't cross S*)
× P(e_2 doesn't cross S* | e_1 doesn't cross S*)
…
× P(e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ doesn't cross S*)



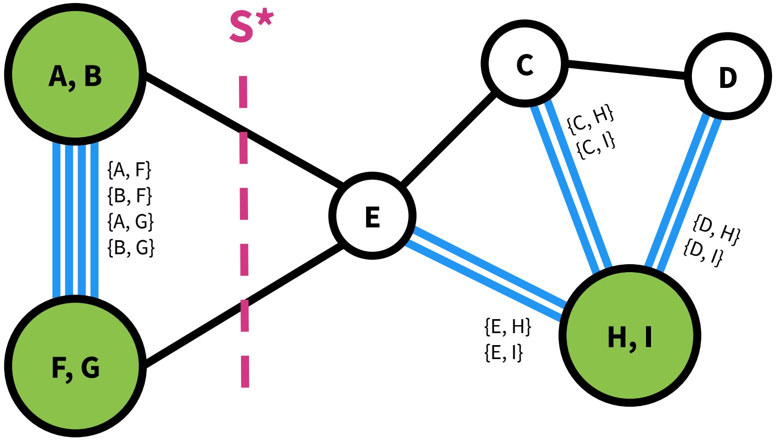
Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?



Karger's Algorithm

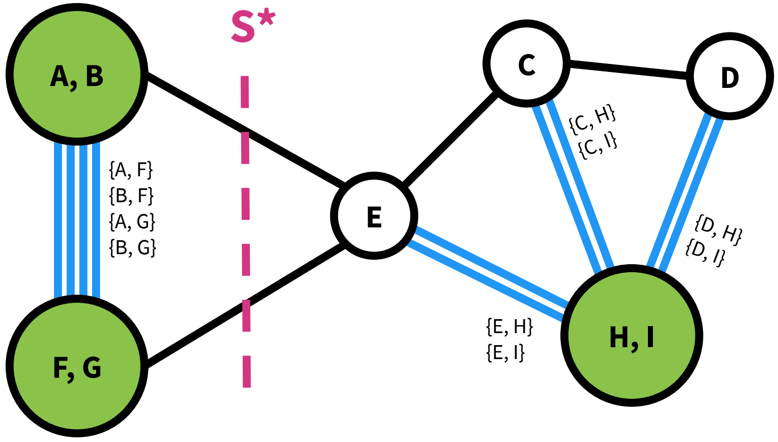
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 \,/\, \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

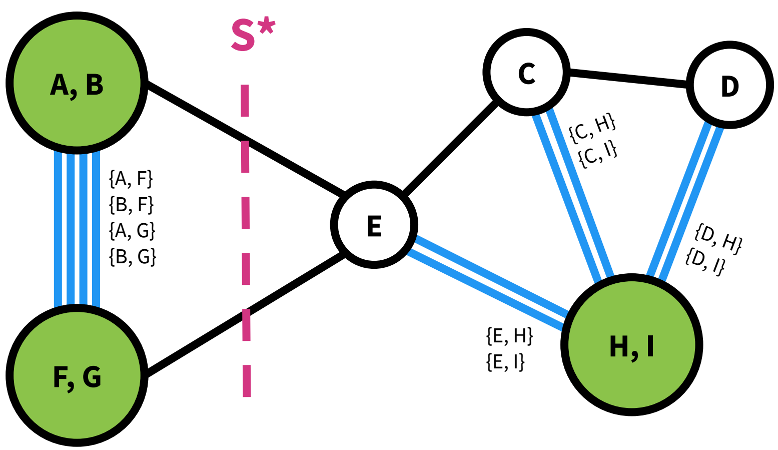
$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

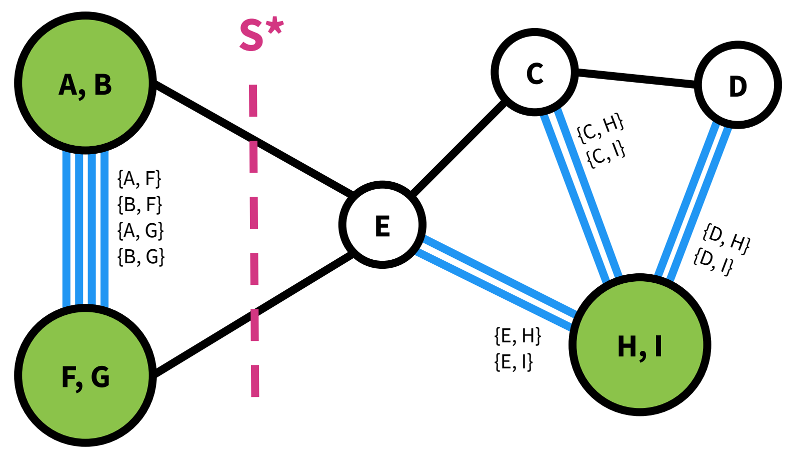
Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).

So there are at least **(n-j+1)k/2** total edges.



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose, after j-1 iterations, `karger` hasn't messed up yet! What's the probability of messing up now?

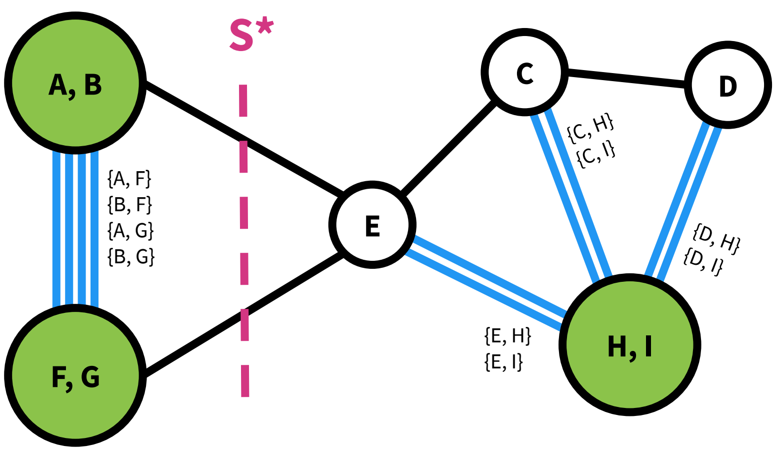
Suppose there are k edges that cross **S***.

All remaining vertices must have degree at least k (otherwise there would be a smaller cut).

So there are at least **(n-j+1)k/2** total edges.

So the probability that `karger` chooses one of the k edges crossing **S*** at step j is at most

$$\frac{k}{\frac{(n-j+1)k}{2}} = \frac{2}{n - j + 1}$$



Karger's Algorithm

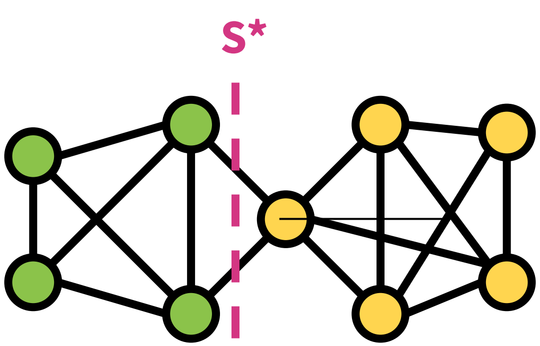
The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose S^* is a min-cut and suppose we select edges e_1, e_2, …, e_{n-2}.

Then P(karger returns S^*) = P(e_1 doesn't cross S^*)
× P(e_2 doesn't cross S^* | e_1 doesn't cross S^*)
…
× P(e_{n-2} doesn't cross S^* | e_1, …, e_{n-3} doesn't cross S^*)



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

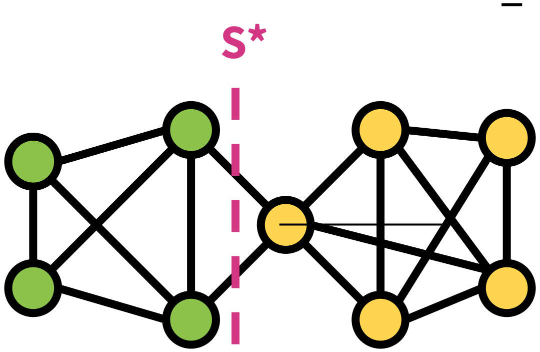
Then P(karger returns S*) = P(e_1 doesn't cross S*)

× P(e_2 doesn't cross S* | e_1 doesn't cross S*)

…

× P(e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ doesn't cross S*)

$$\geq \frac{(n-2)}{n} \; \frac{(n-3)}{(n-1)} \; \frac{(n-4)}{(n-2)} \; \frac{(n-5)}{(n-3)} \; \frac{(n-6)}{(n-4)} \cdots \frac{4}{6} \; \frac{3}{5} \; \frac{2}{4} \; \frac{1}{3}$$



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 \Big/ \binom{n}{2}$$

Proof, cont.:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

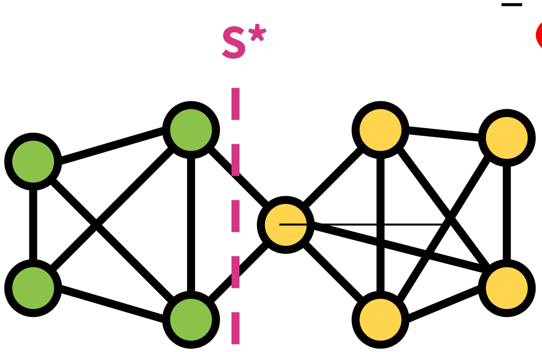
Then P(karger returns S*) = P(e_1 doesn't cross S*)

$\times$ P(e_2 doesn't cross S* | e_1 doesn't cross S*)

…

$\times$ P(e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ doesn't cross S*)

$$\geq \frac{(n-2)}{n} \frac{(n-3)}{(n-1)} \frac{(n-4)}{(n-2)} \frac{(n-5)}{(n-3)} \frac{(n-6)}{(n-4)} \cdots \frac{4}{6} \frac{3}{5} \frac{2}{4} \frac{1}{3}$$



Karger's Algorithm

The probability that Karger's algorithm returns a minimum cut is …

$$\geq 1 / \binom{n}{2}$$

Proof, cont.:

Suppose S* is a min-cut and suppose we select edges $e_1, e_2, \ldots, e_{n-2}$.

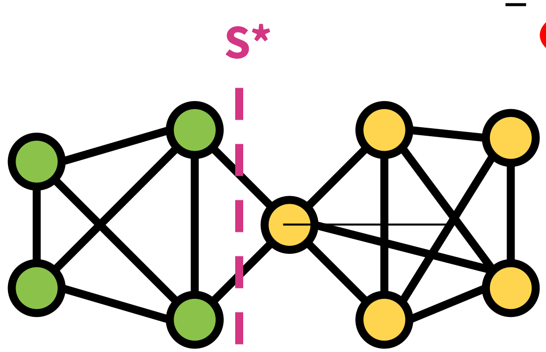
Then P(karger returns S*) = P(e_1 doesn't cross S*)

× P(e_2 doesn't cross S* | e_1 doesn't cross S*)

…

× P(e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ doesn't cross S*)

$$\geq \frac{(n-2)}{n}\,\frac{(n-3)}{(n-1)}\,\frac{(n-4)}{(n-2)}\,\frac{(n-5)}{(n-3)}\,\frac{(n-6)}{(n-4)}\cdots\frac{4}{6}\,\frac{3}{5}\,\frac{2}{4}\,\frac{1}{3}$$



= 2/(n(n-1))

= 1/(nC2)

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?
- Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
  P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

Karger's Algorithm

1/(nC2) isn't all that great …

- For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

- How many times T do we need to repeat `karger` to obtain this probability?
- Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
  P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)
- P(find the min-cut after T times) ≥ 0.9

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$

Karger's Algorithm

1/(nC2) isn't all that great …

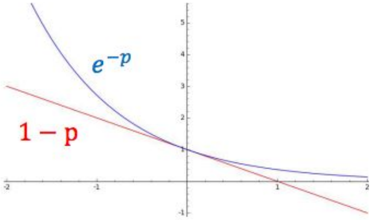
For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$



$1 - p \leq e^{-p}$

Karger's Algorithm

1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

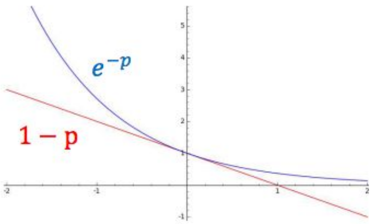
Suppose we want to find the min-cut with probability 0.9. What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)

P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

T = (nC2) ln (1/0.1) times



$1 - p \leq e^{-p}$

Karger's Algorithm

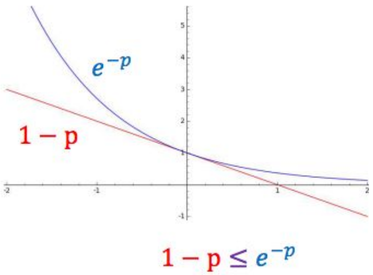
1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

T = (nC2) ln (1/0.1) times

Suppose we want to find the min-cut with probability p.

Karger's Algorithm

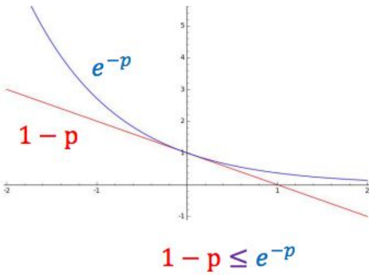
1/(nC2) isn't all that great …

For our example of n = 9, 1/(9C2) = 0.028.

Suppose we want to find the min-cut with probability 0.9.
What can we do? 🤔

How many times T do we need to repeat `karger` to obtain this probability?

Note that if P(find the min-cut after 1 time) ≥ 1/(nC2), then
P(don't find the min-cut after 1 time ≤ 1 - 1/(nC2)



P(find the min-cut after T times) ≥ 0.9
⇔ P(don't find the min-cut after T times) ≤ 0.1.
P(don't find the min-cut after T times) = $(1 - 1/(nC2))^T$
$\leq (e^{-1/(nC2)})^T = 0.1$

T = (nC2) ln (1/0.1) times

Suppose we want to find the min-cut with probability p.
Then we must repeat karger **T = (nC2) ln (1/(1-p)) times**.

Karger's Algorithm

T = (nC2) ln (1/(1-p)) times = $O(|V|^2)$ times, so the overall runtime is $O(|V|^4)$.

- Treating 1-p as a constant.
- If we use union-find data structures, then we can do better.
- This might seem lousy, but then consider that enumerating over all possible cuts to find the min-cut requires $O(2^{|V|})$.
- This is a huge improvement!

Karger's Algorithm

```
algorithm karger_loop(G=(V,E), threshold):
  cur_min_cut = None
  n = V.length, p = threshold
  for t = 1 to (nC2)ln(1/(1-p)) :
    candidate_cut = karger(G)
    if candidate_cut.size < cur_min_cut.size:
      cur_min_cut = candidate_cut
  return cur_min_cut
```

Runtime: $O(|V|^4)$

Karger’s Algorithm

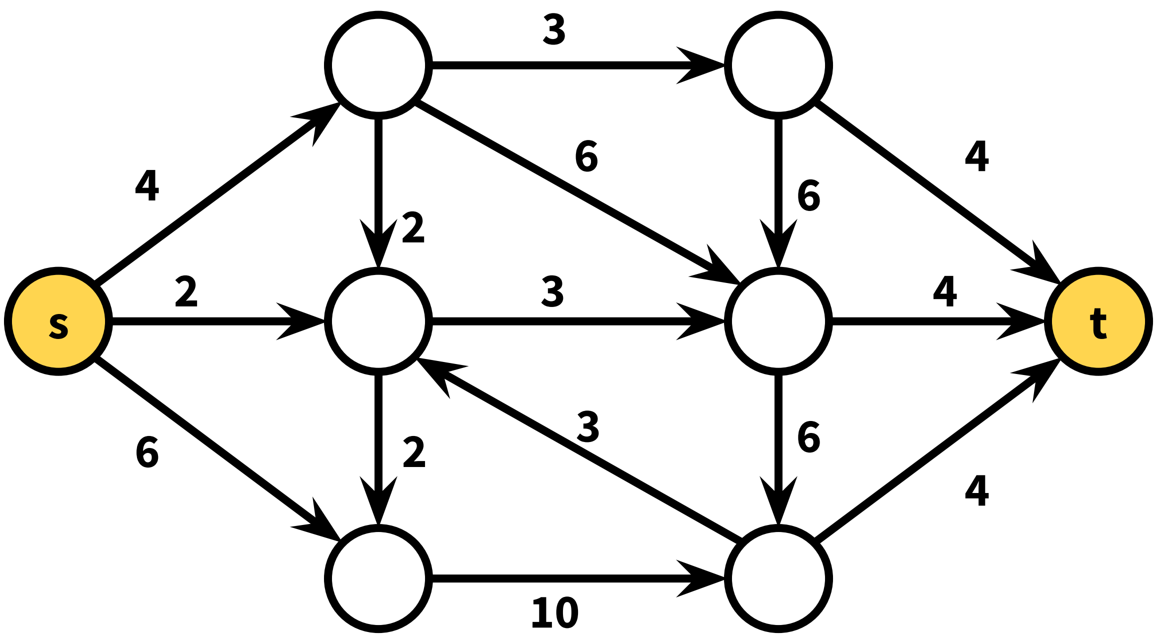
Upshot: Whenever we have a Monte-Carlo algorithm with a small probability of success, we can boost the probability of success by repeating it a bunch of times and taking the best solution!

Ford-Fulkerson Algorithm

Minimum Cuts

Each edge has a flow.

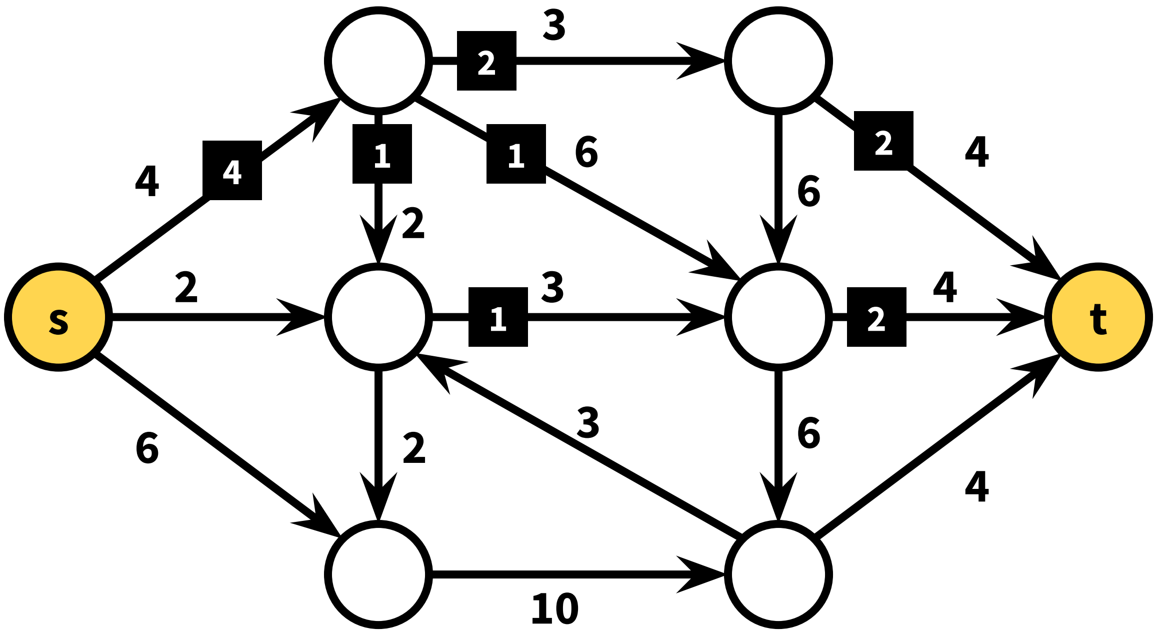
The flow on an edge must be less than its capacity and at each vertex, the incoming flows must equal the outgoing flows.



Minimum Cuts

Each edge has a flow.

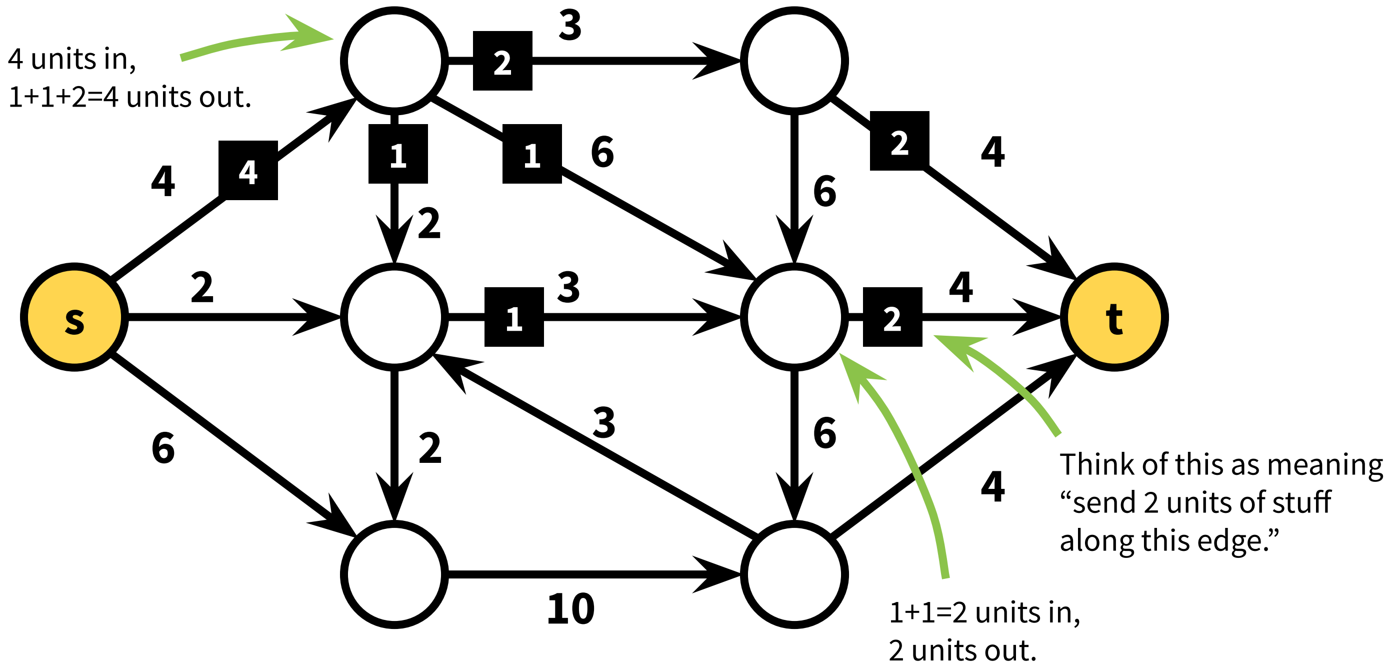
- The flow on an edge must be less than its capacity and at each vertex, the incoming flows must equal the outgoing flows.



Minimum Cuts

Each edge has a flow.

The flow on an edge must be less than its capacity and at each vertex, the incoming flows must equal the outgoing flows.

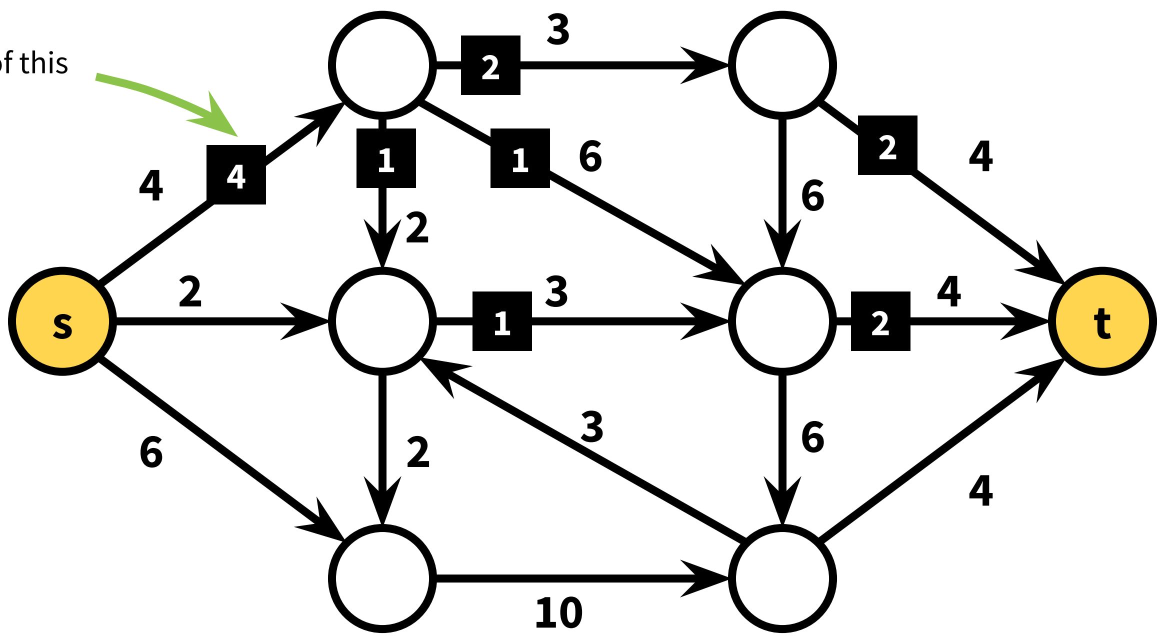


Minimum Cuts

Each edge has a flow.

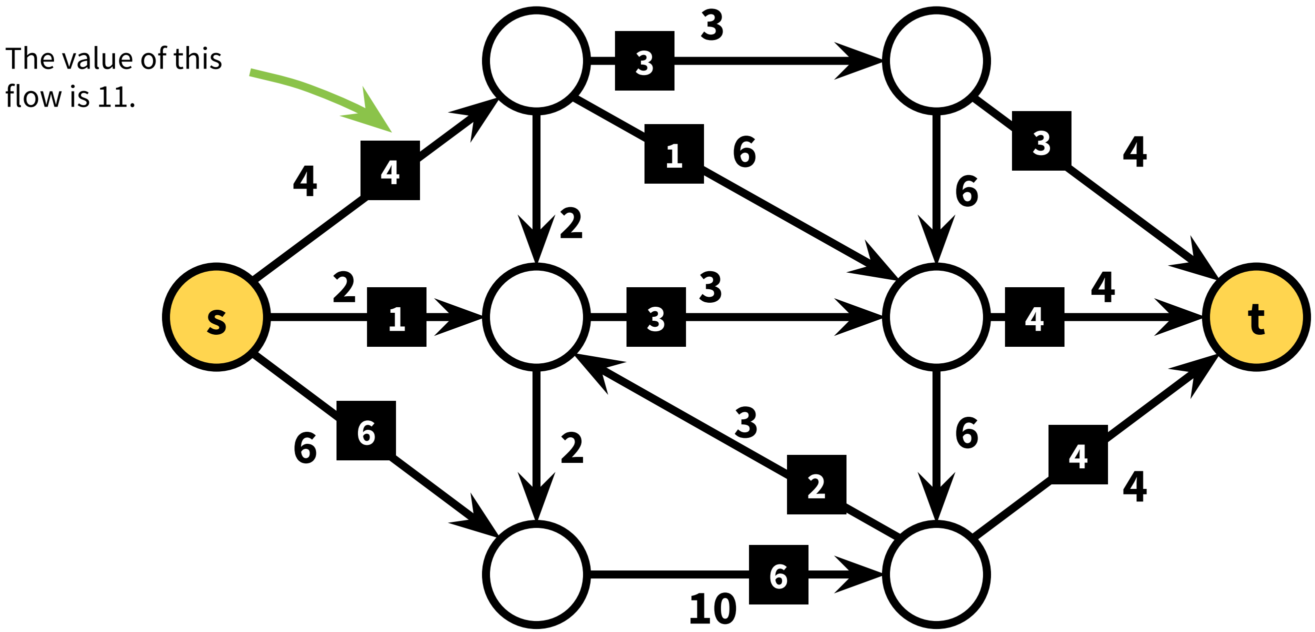
The value of a flow is the amount of stuff coming out of s and the amount of stuff going into t. Due to conservation of flows at vertices, these are equal.

The value of this flow is 4.



Minimum Cuts

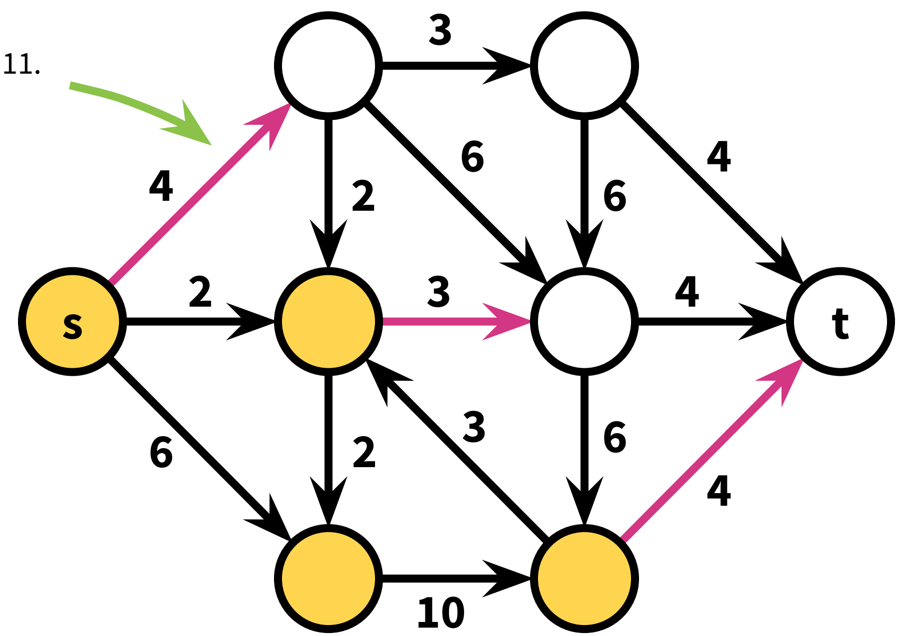
A maximum flow is a flow of maximum value.



Minimum Cuts

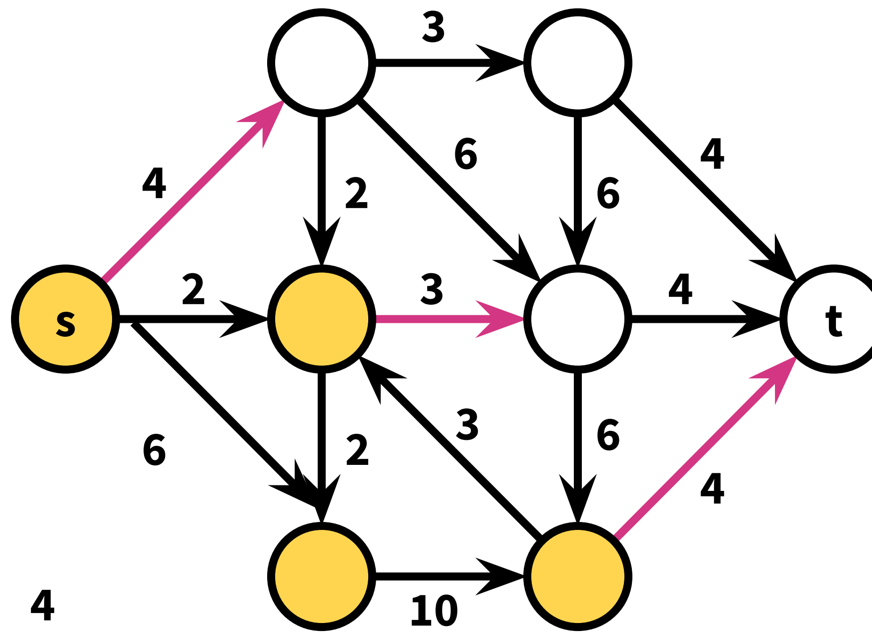
A minimum s-t cut is a cut which separates s from t with minimum capacity.

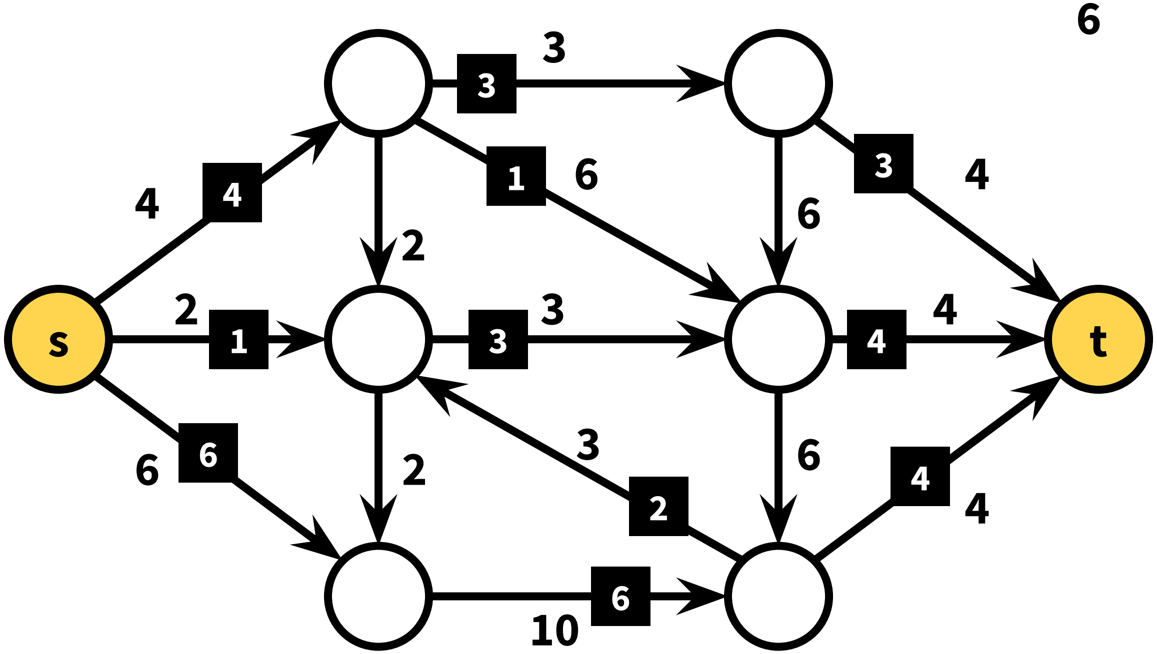
The cost of this cut is 11.



Max-Flow Min-Cut Theorem

This isn’t a coincidence!





Max-Flow Min-Cut Theorem

The value of a max-flow from s to t is equal to the cost of a min s-t cut.

Intuition: in a max-flow, the min-cut “fills-up,” and this is the bottleneck.

Max-Flow Min-Cut Theorem

Lemma 1: max flow ≤ min cut

Proof by picture

Lemma 2: max flow ≥ min cut

Proof by algorithm, using a "residual graph" G_f

Sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we do left implication:

Claim: If there is a path from s to t in G_f, then we can increase the flow in G.

Hence we couldn't have started with a max flow.

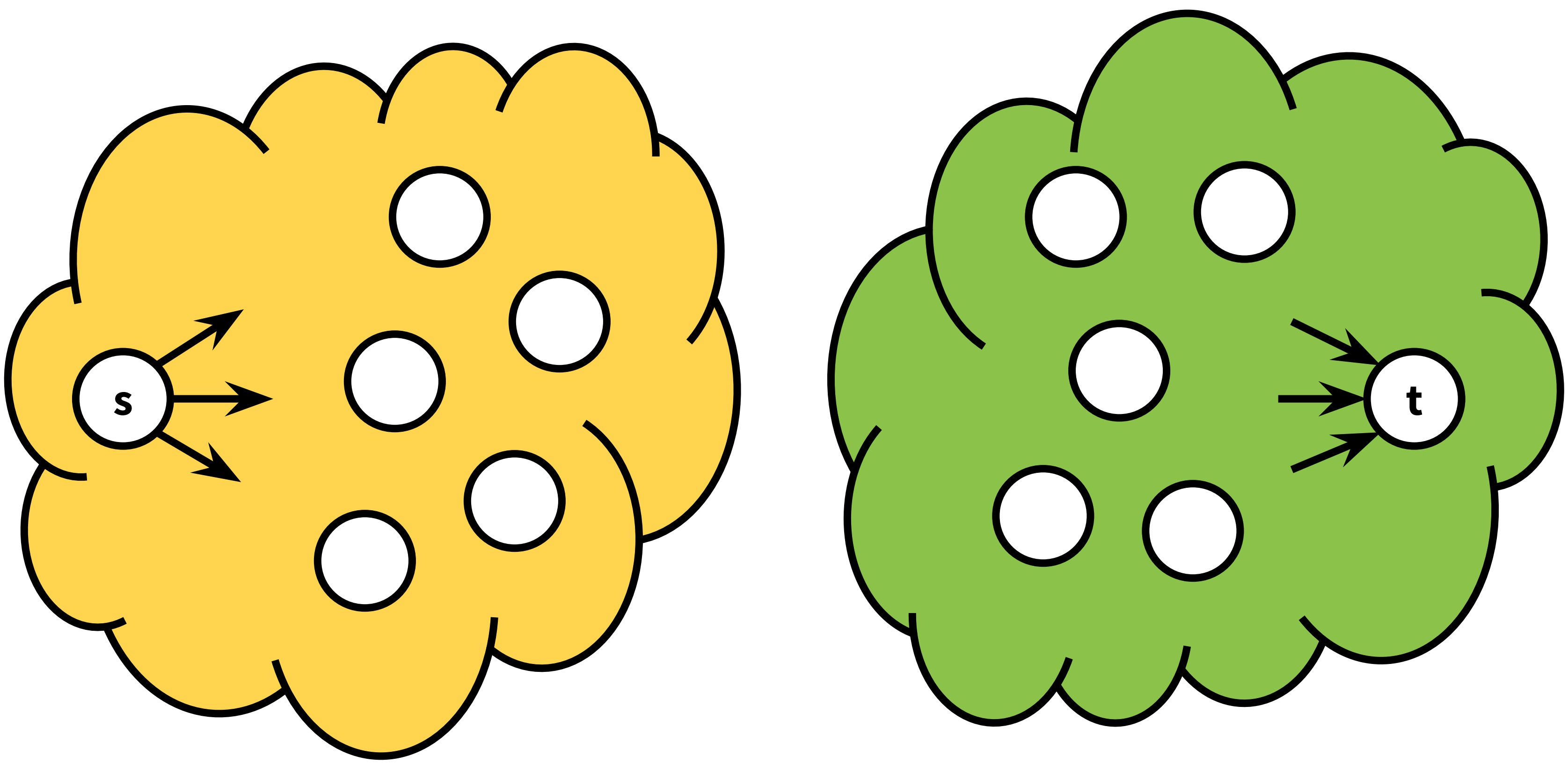
Proof by picture for right implication

This claim gives us an algorithm: Find paths from s to t in G_f and keep increasing the flow until you can't anymore.

Max-Flow Min-Cut Theorem

Lemma 1: max flow ≤ min cut

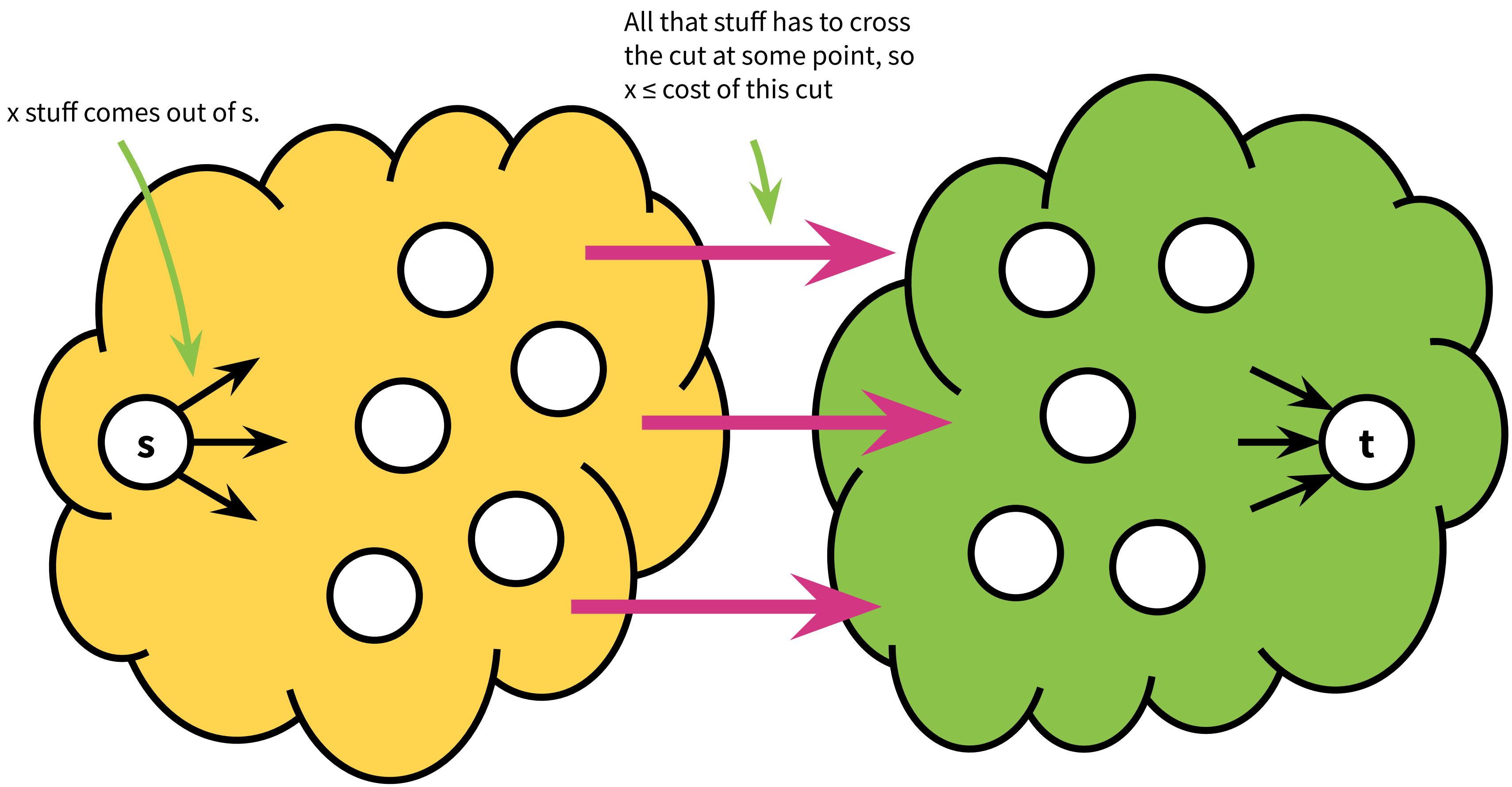
For ANY s-t flow and ANY s-t cut, the value of the flow is at most the cost of the cut.



Max-Flow Min-Cut Theorem

Lemma 1: max flow ≤ min cut

For ANY s-t flow and ANY s-t cut, the value of the flow is at most the cost of the cut.

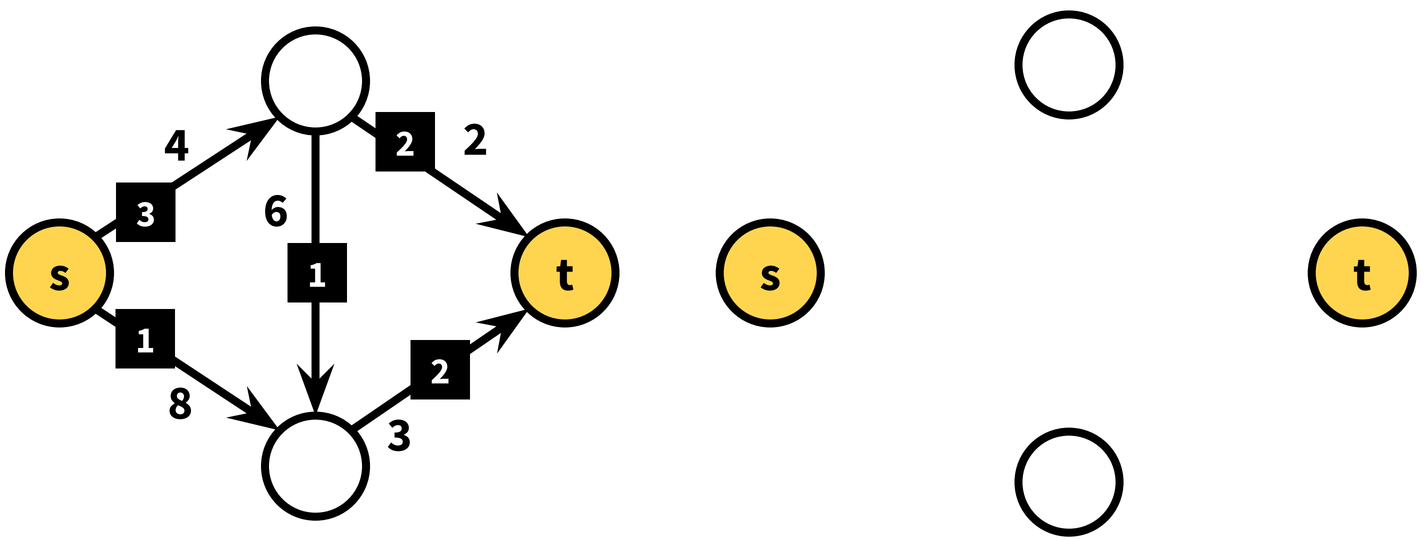


Ford-Fulkerson Algorithm

- Start with zero flow
- We will maintain a residual graph G_f
- A path from s to t in G_f will give us a way to improve our flow.
- We will continue until there are no s-t paths left.

Max-Flow Min-Cut Theorem

We can create a residual graph G_f from a flow.



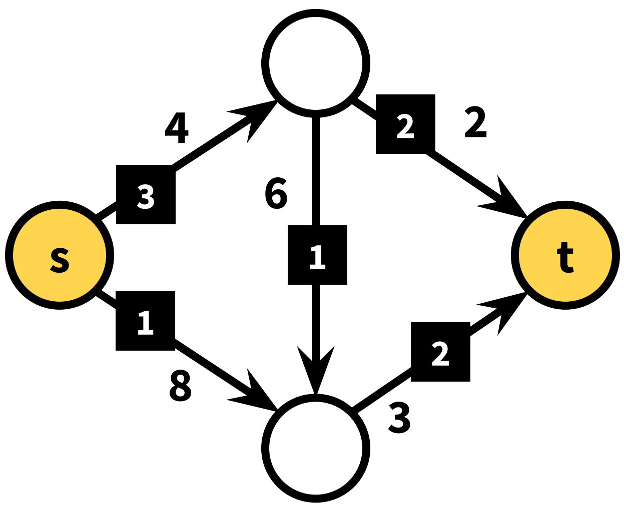
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ \ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & else \end{cases}$$

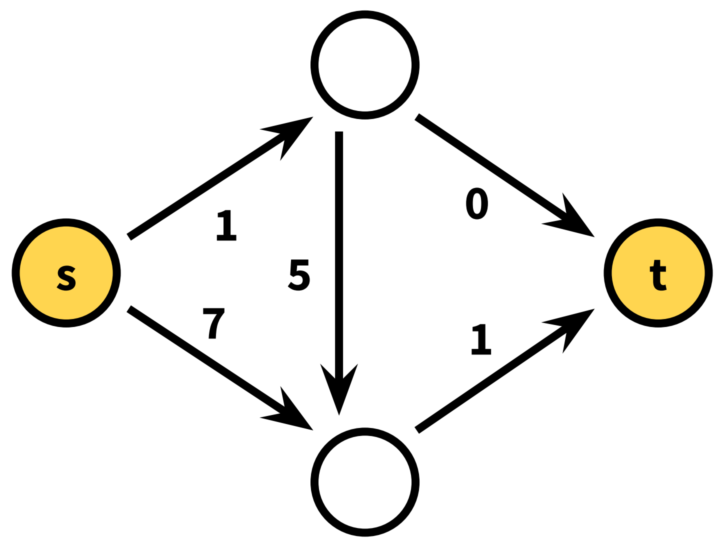
f(u, v) is the flow on edge (u, v)
c(u, v) is the capacity on edge (u, v)

Max-Flow Min-Cut Theorem

We can create a residual graph G_f from a flow.

Forward edges are the amount that's left.





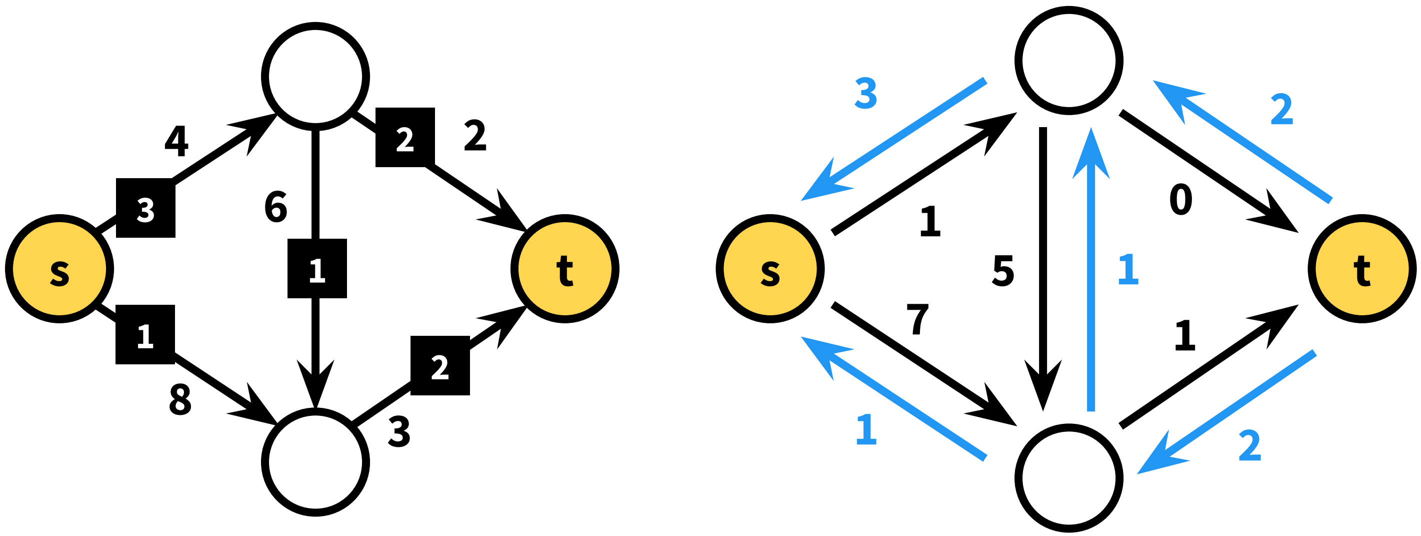
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if \ \ (u,v) \in E \\ f(v,u) & if \ (v,u) \in E \\ 0 & else \end{cases}$$

f(u, v) is the flow on edge (u, v)
c(u, v) is the capacity on edge (u, v)

Max-Flow Min-Cut Theorem

We can create a residual graph G_f from a flow.

Forward edges are the amount that's left. Backward edges are the amount that's been used.



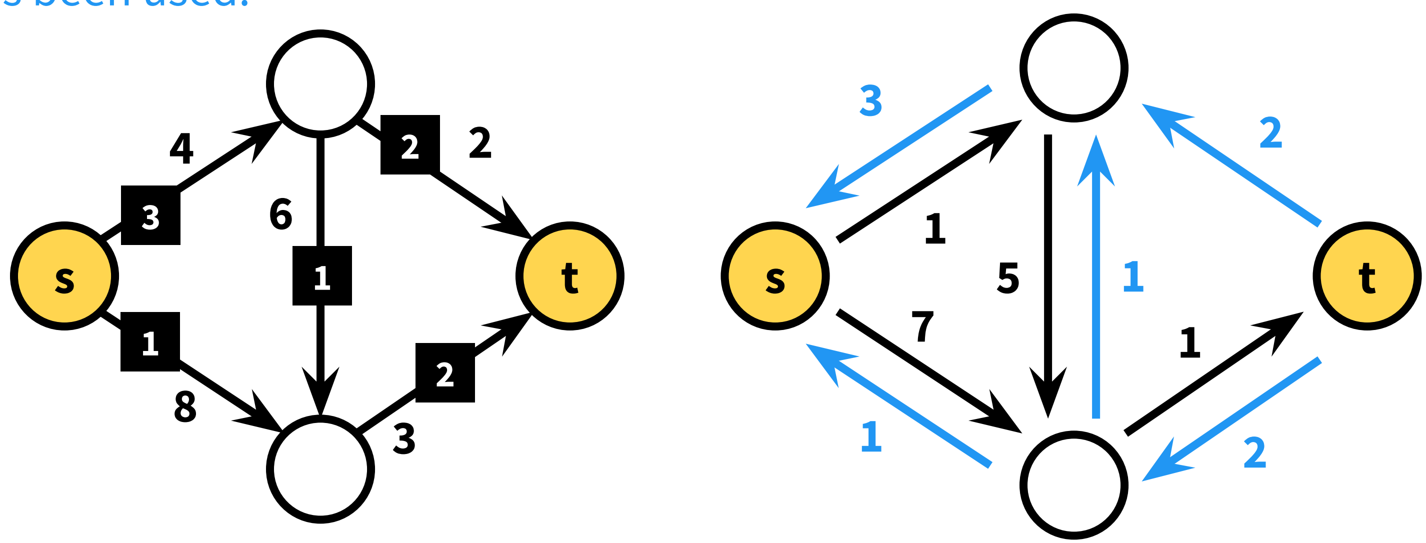
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ \ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & else \end{cases}$$

f(u, v) is the flow on edge (u, v)
c(u, v) is the capacity on edge (u, v)

Max-Flow Min-Cut Theorem

We can create a residual graph G_f from a flow.

Forward edges are the amount that's left. Backward edges are the amount that's been used.

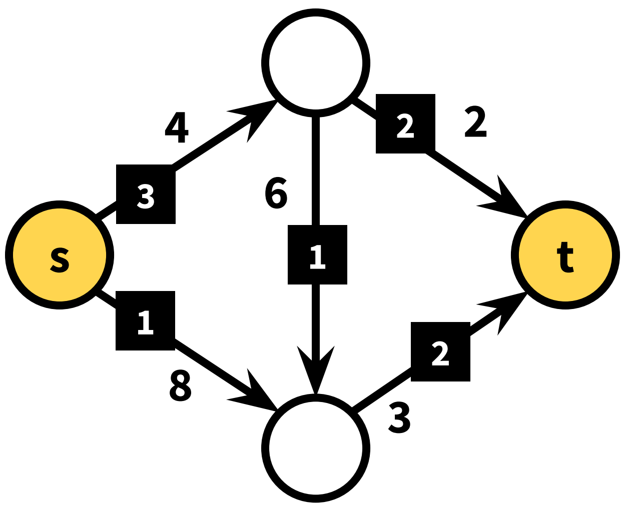


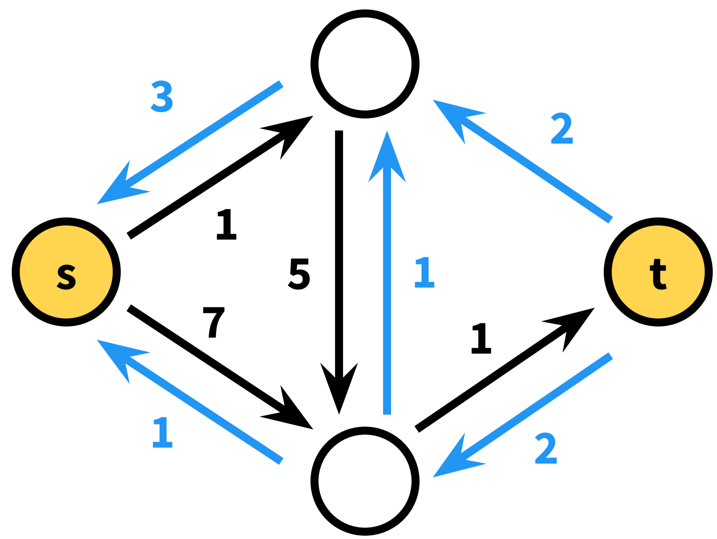
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ \ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & else \end{cases}$$

f(u, v) is the flow on edge (u, v)
c(u, v) is the capacity on edge (u, v)

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

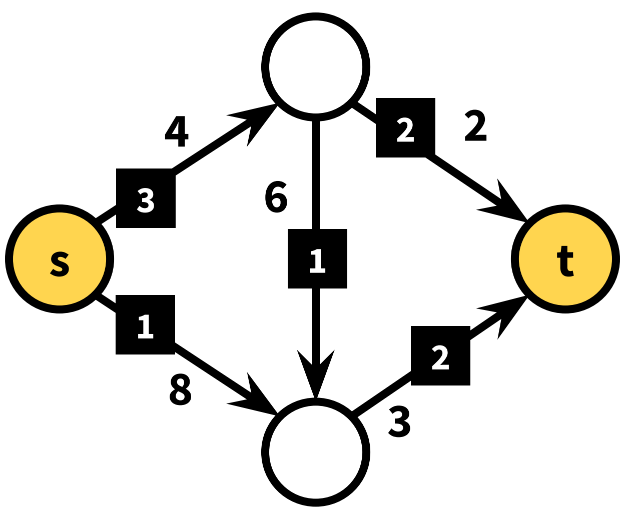


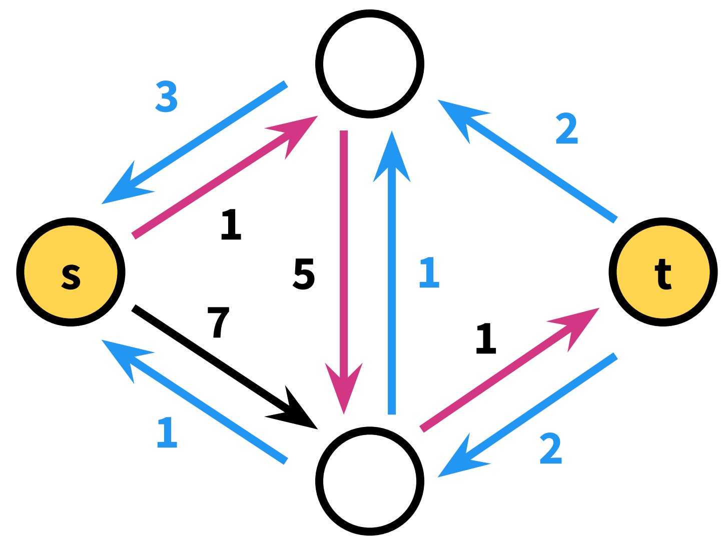


Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

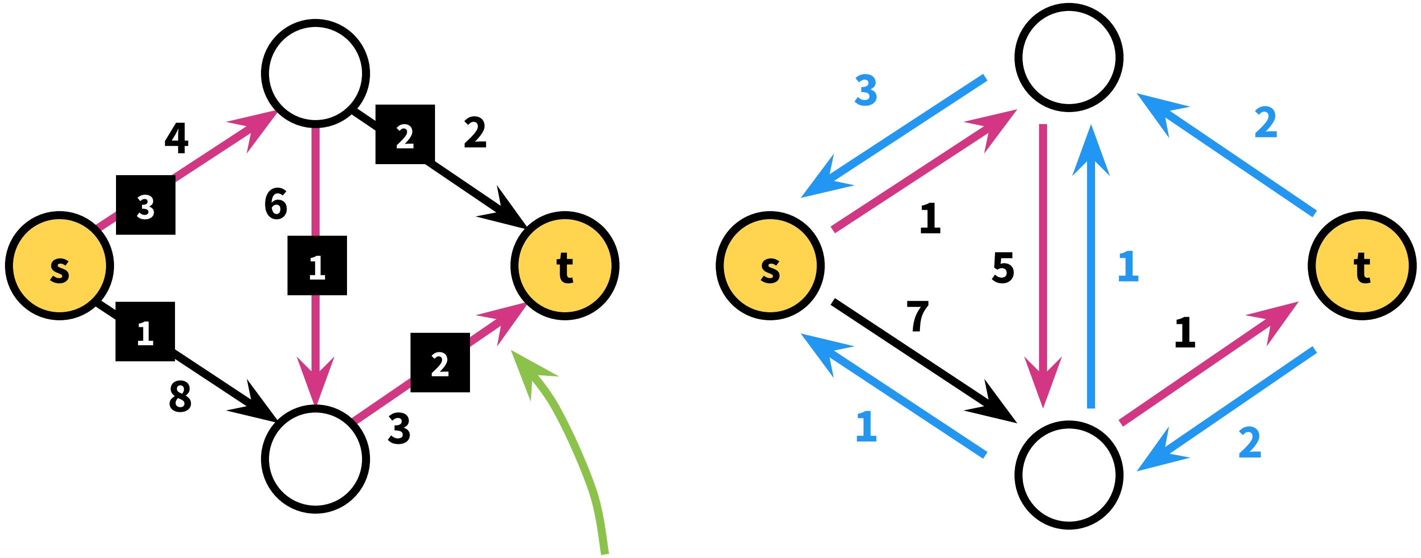
e.g. **t is reachable from s** in G_f (on the right), so not a max flow (on the left).





Max-Flow Min-Cut Theorem

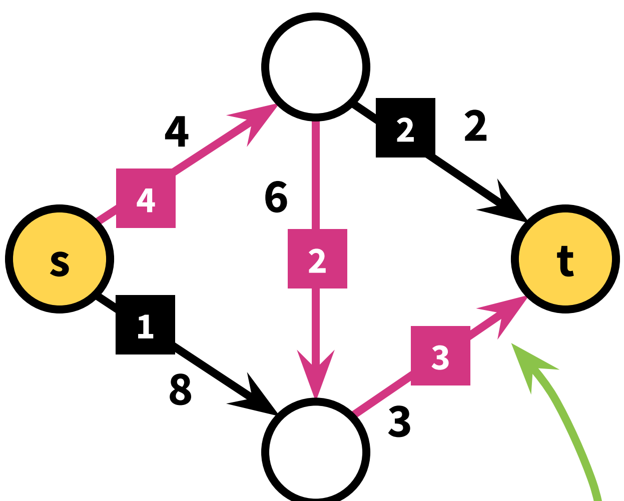
Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.



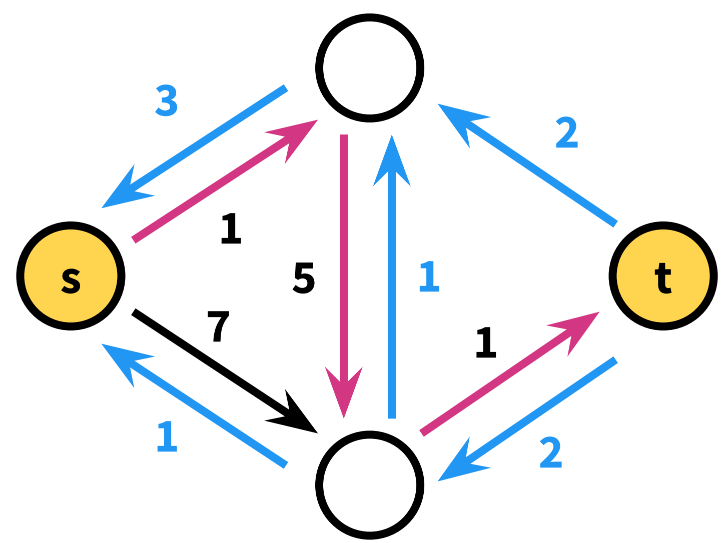
To see that this flow is not maximal, notice that we can improve it by sending one more unit more stuff along this path

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

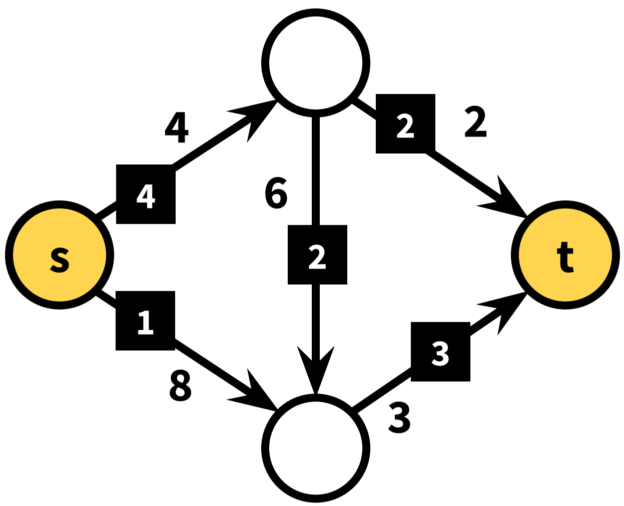


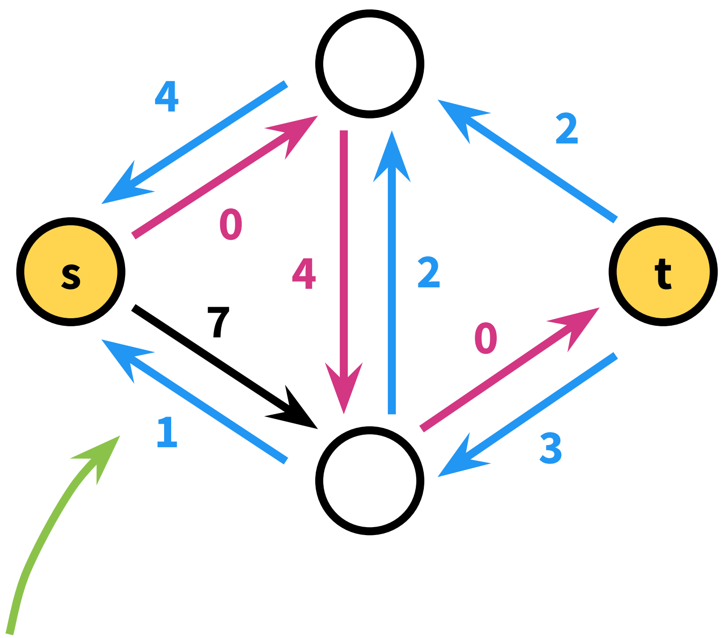
To see that this flow is not maximal, notice that we can improve it by sending one more unit more stuff along this path



Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.





Then update G_f

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we prove the left implication (if f is a max flow, t is not reachable from s in G_f) by proving the contrapositive.

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we prove the left implication (if f is a max flow, t is not reachable from s in G_f) by proving the contrapositive: **If t is reachable from s in G_f then f is not a max flow.**

Suppose there is a path from s to t in G_f (called an augmenting path).

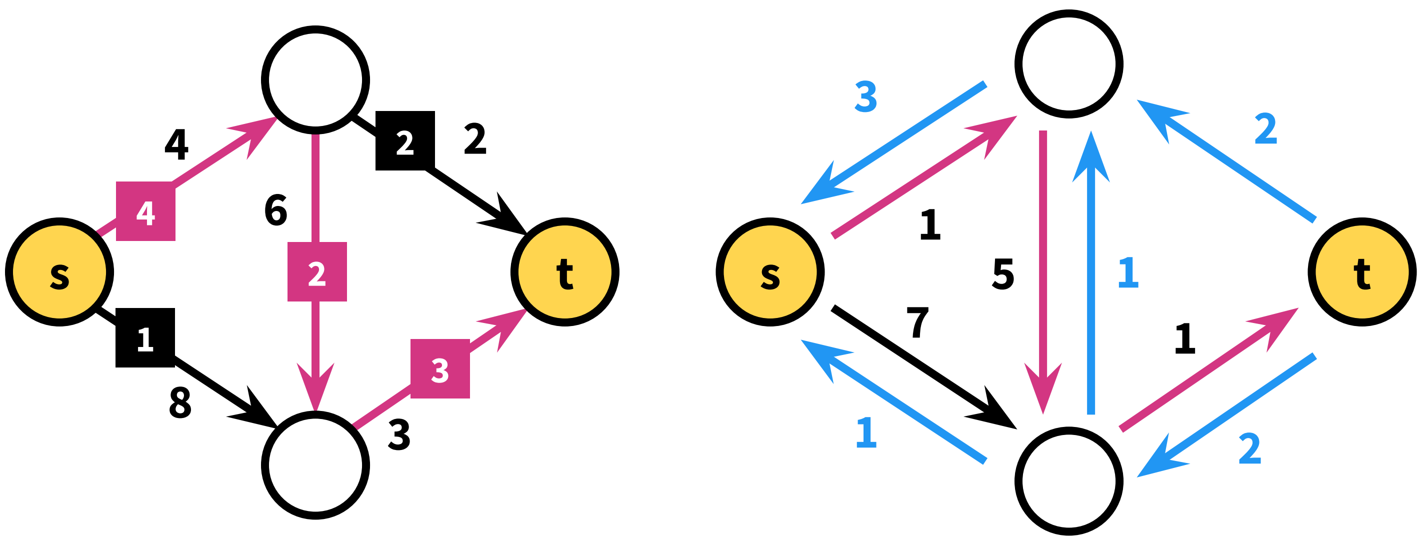
Claim: If there is an augmenting path, we can increase the flow along that path.

Augmenting along this path and updating the flow results in a bigger flow, so we couldn't have started with a max flow.

Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

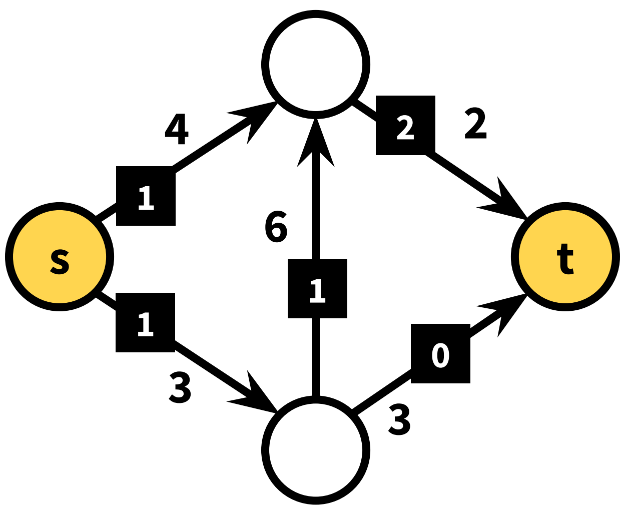


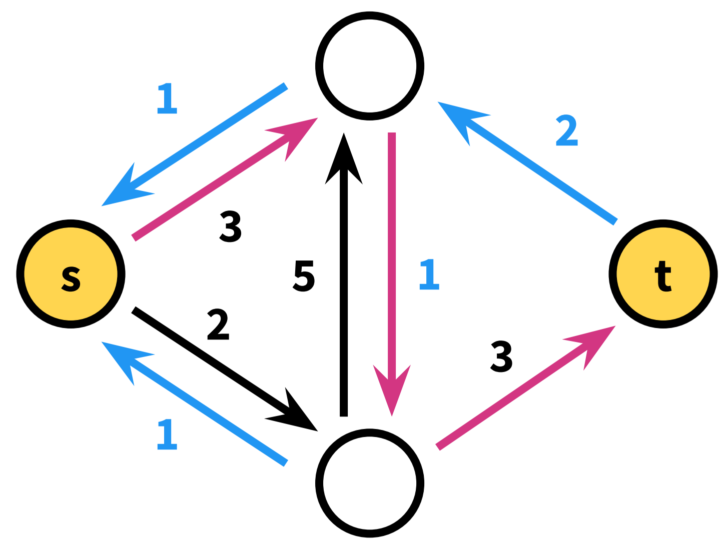
Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).





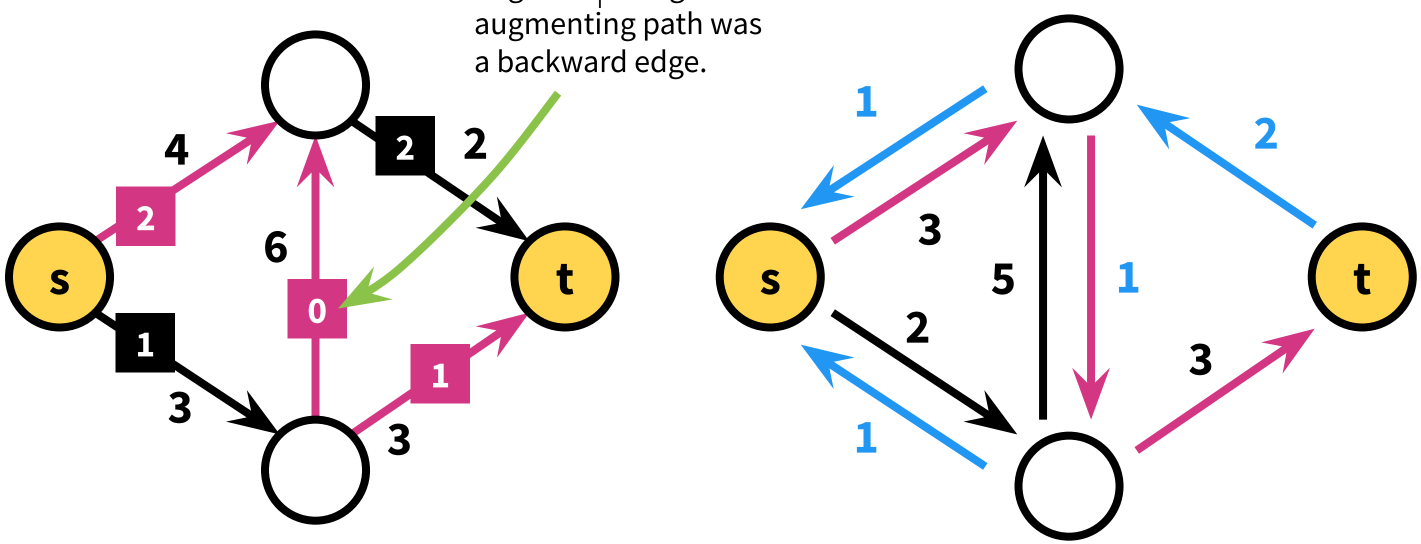
Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).

We subtract flow here; the corresponding edge in G_f along the augmenting path was a backward edge.

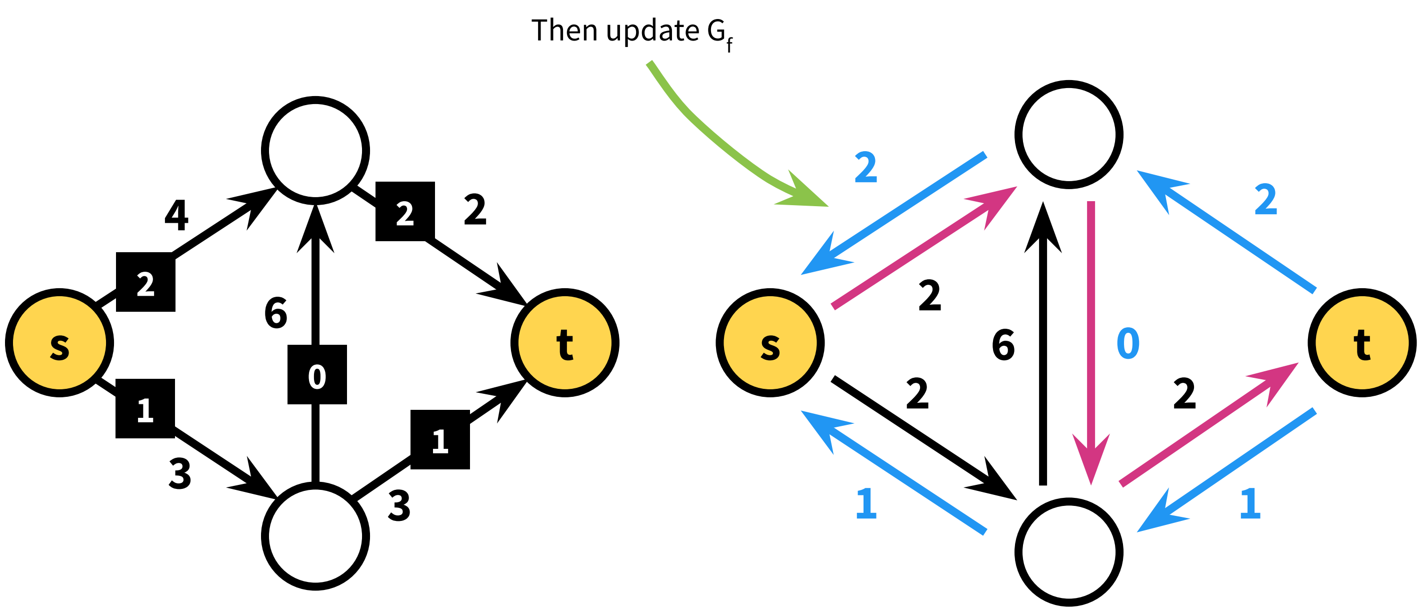


Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).



Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).

To handle both cases, we can follow the following procedure:

- x = min weight on any edge in P from G_f

Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).

To handle both cases, we can follow the following procedure:

- x = min weight on any edge in P from G_f
- for (u, v) in P:
  - If (u, v) in E, $f_{new}(u, v) = f(u, v) + x$
  - If (v, u) in E, $f_{new}(u, v) = f(u, v) + x$

Max-Flow Min-Cut Theorem

Claim: If there is an augmenting path, we can increase the flow along that path.

Case 1: Every edge is a forward edge, so we can just increase the flow on all edges (this is the case we just saw).

Case 2: Some edges are backward edges on the path (a slightly different example below).

To handle both cases, we can follow the following procedure:

- x = min weight on any edge in P from G_f
- for (u, v) in P:
  - If (u, v) in E, $f_{new}(u, v) = f(u, v) + x$
  - If (v, u) in E, $f_{new}(u, v) = f(u, v) + x$
- return f_{new}

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we prove the left implication (if f is a max flow, t is not reachable from s in G_f) by proving the contrapositive: **If t is reachable from s in G_f then f is not a max flow.**

Suppose there is a path from s to t in G_f (called an augmenting path).

Claim: If there is an augmenting path, we can increase the flow along that path.

Augmenting along this path and updating the flow results in a bigger flow, so we couldn't have started with a max flow.

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we prove the left implication (if f is a max flow, t is not reachable from s in G_f) by proving the contrapositive: **If t is reachable from s in G_f then f is not a max flow.** **DONE!**

Suppose there is a path from s to t in G_f (called an augmenting path).

Claim: If there is an augmenting path, we can increase the flow along that path. **DONE!**

Augmenting along this path and updating the flow results in a bigger flow, so we couldn't have started with a max flow.

Max-Flow Min-Cut Theorem

Lemma 2 sub-lemma: t is not reachable from s in G_f iff f is a max flow.

First we prove the left implication (if f is a max flow, t is not reachable from s in G_f) by proving the contrapositive: **If t is reachable from s in G_f then f is not a max flow.** **DONE!**

Suppose there is a path from s to t in G_f (called an augmenting path).

Claim: If there is an augmenting path, we can increase the flow along that path. **DONE!**

Augmenting along this path and updating the flow results in a bigger flow, so we couldn't have started with a max flow.

Now we prove the right implication: **If t is not reachable from s in G_f then f is a max flow.**

Max-Flow Min-Cut Theorem

Claim: If t is not reachable from s in G_f then f is a max flow.

- Consider the cut **{things reachable from s}**, **{things not reachable from s}**
- The flow from s to t is equal to the cost of this cut.
  - Similar to the proof-by-picture from before
    - All of the stuff has to cross the cut
    - The edges in the cut are **full** because they don't exist in G_f

Max-Flow Min-Cut Theorem

Claim: If t is not reachable from s in G_f then f is a max flow.

Consider the cut {things reachable from s}, {things not reachable from s}

The flow from s to t is equal to the cost of this cut.

Similar to the proof-by-picture from before

All of the stuff has to cross the cut

The edges in the cut are **full** because they don't exist in G_f

Thus: this flow value = cost of the cut ≥ min cut ≥ max flow

From above

By definition of min cut being minimum

Lemma 1

Max-Flow Min-Cut Theorem

Claim: If t is not reachable from s in G_f then f is a max flow.

Consider the cut {things reachable from s}, {things not reachable from s}

The flow from s to t is equal to the cost of this cut.

Similar to the proof-by-picture from before

All of the stuff has to cross the cut

The edges in the cut are **full** because they don't exist in G_f

Thus: this flow value = cost of the cut ≥ min cut ≥ max flow

From above

By definition of min cut being minimum

Lemma 1

By definition of max flow being maximum: max flow ≥ this flow value, so they must be equal!

Ford-Fulkerson Algorithm

```
def ford_fulkerson(G=(V,E)):
  f = zero flow
  G_f = G
  while t is reachable from s in G_f:
    find path P from s to t in G_f  # e.g. BFS
    f = increase_flow(P, f)
    update G_f
  return f
```

Runtime: $O(|V||E|^2)$