CSE100: Design and Analysis of Algorithms
Lecture 25 – Min Cut and Karger's Algorithm

Apr 26th 2022

Min Cut, Karger and Karger-Stein's Algorithms

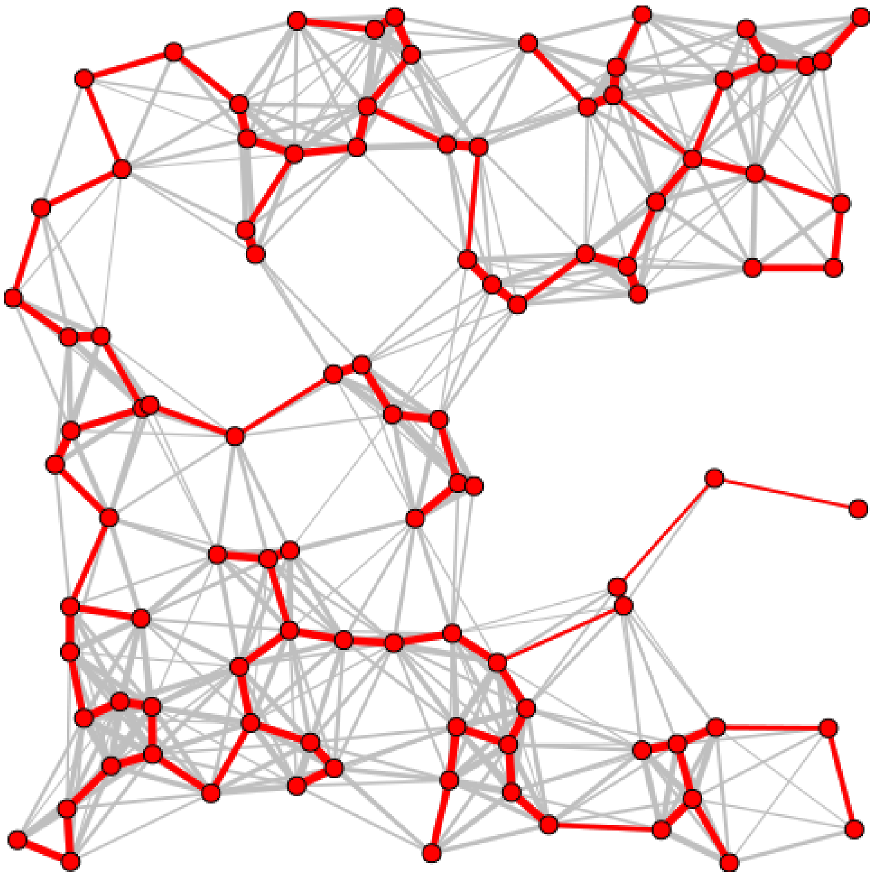
Question from last time

- Does Prim's algorithm work with negative edge weights?
  - After all, it looks a lot like Dijkstra…
- Answer is yes! Prim works fine with negative edge weights.
  - To convince yourself, go through the proof and make sure it still works.
  - (Where did we use the fact that the weights were non-negative for Dijkstra?)

Answer – Several places. See proof of Claim 2: When vertex v is marked sure d[v] = d(s, v).

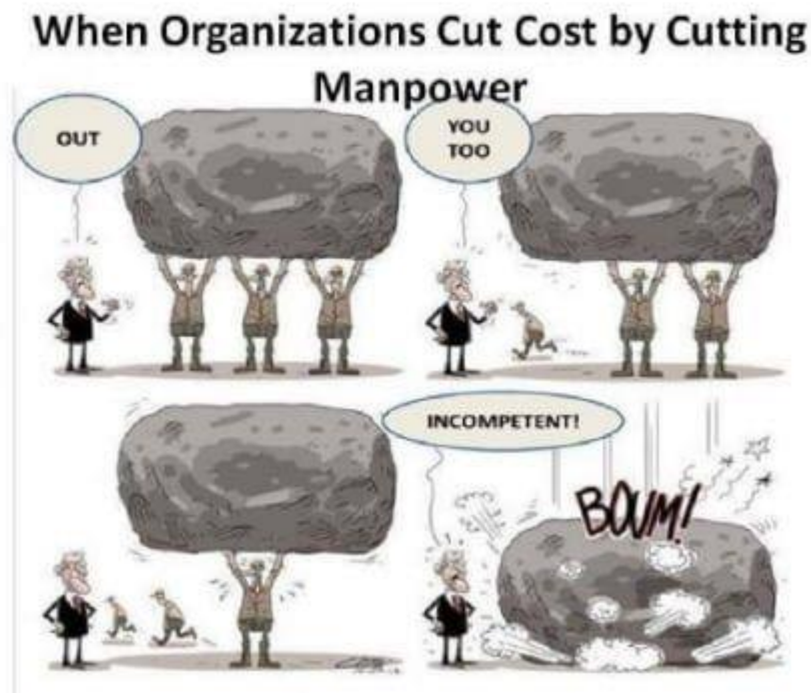
Last time

- Minimum Spanning Trees!
  - Prim's Algorithm
  - Kruskal's Algorithm



Today

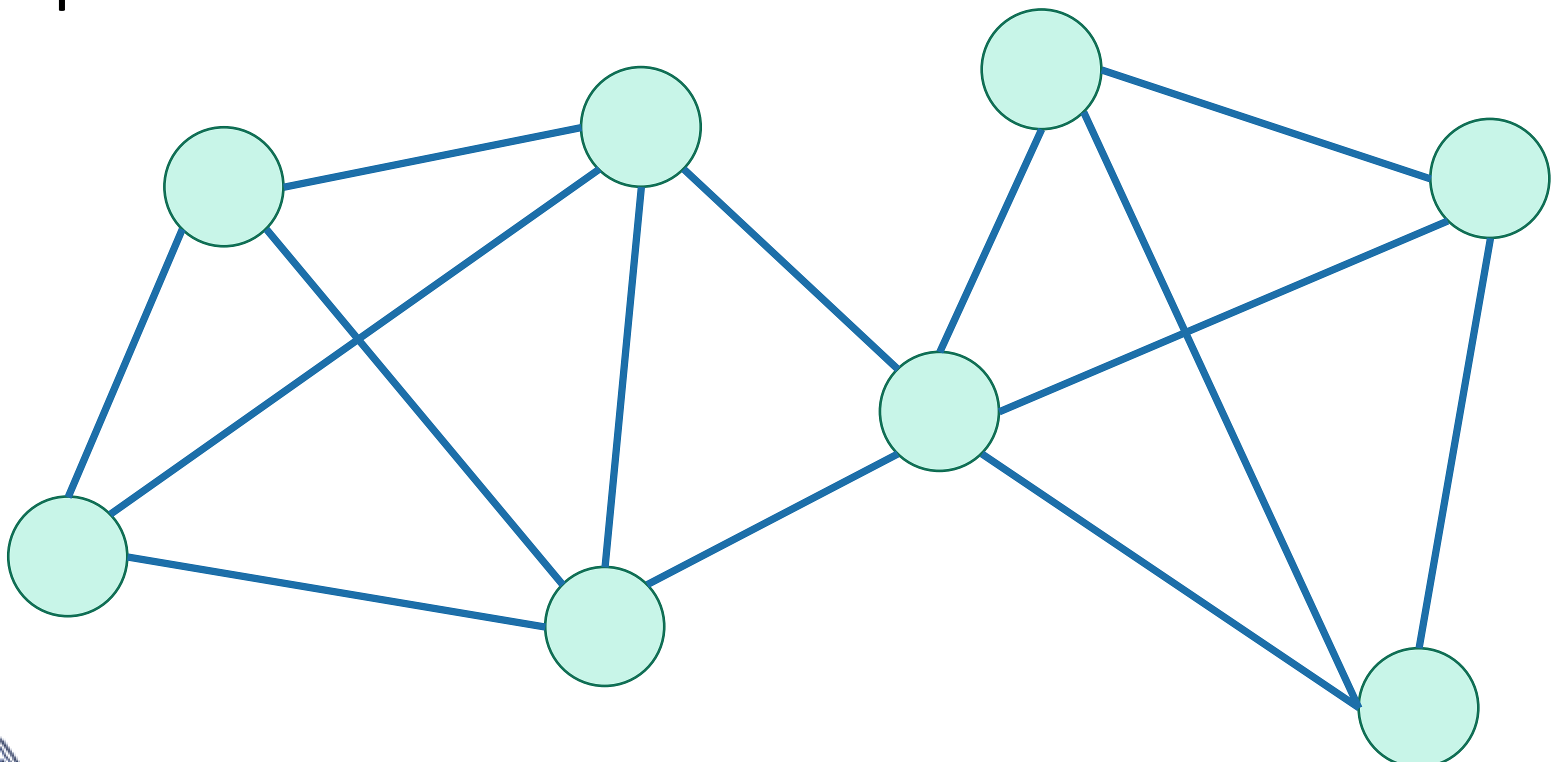
- Minimum Cuts!
  - Karger's algorithm
  - Karger-Stein algorithm



  - Back to **randomized algorithms!**
    - but in a different way than we've seen so far

Recall: cuts in graphs

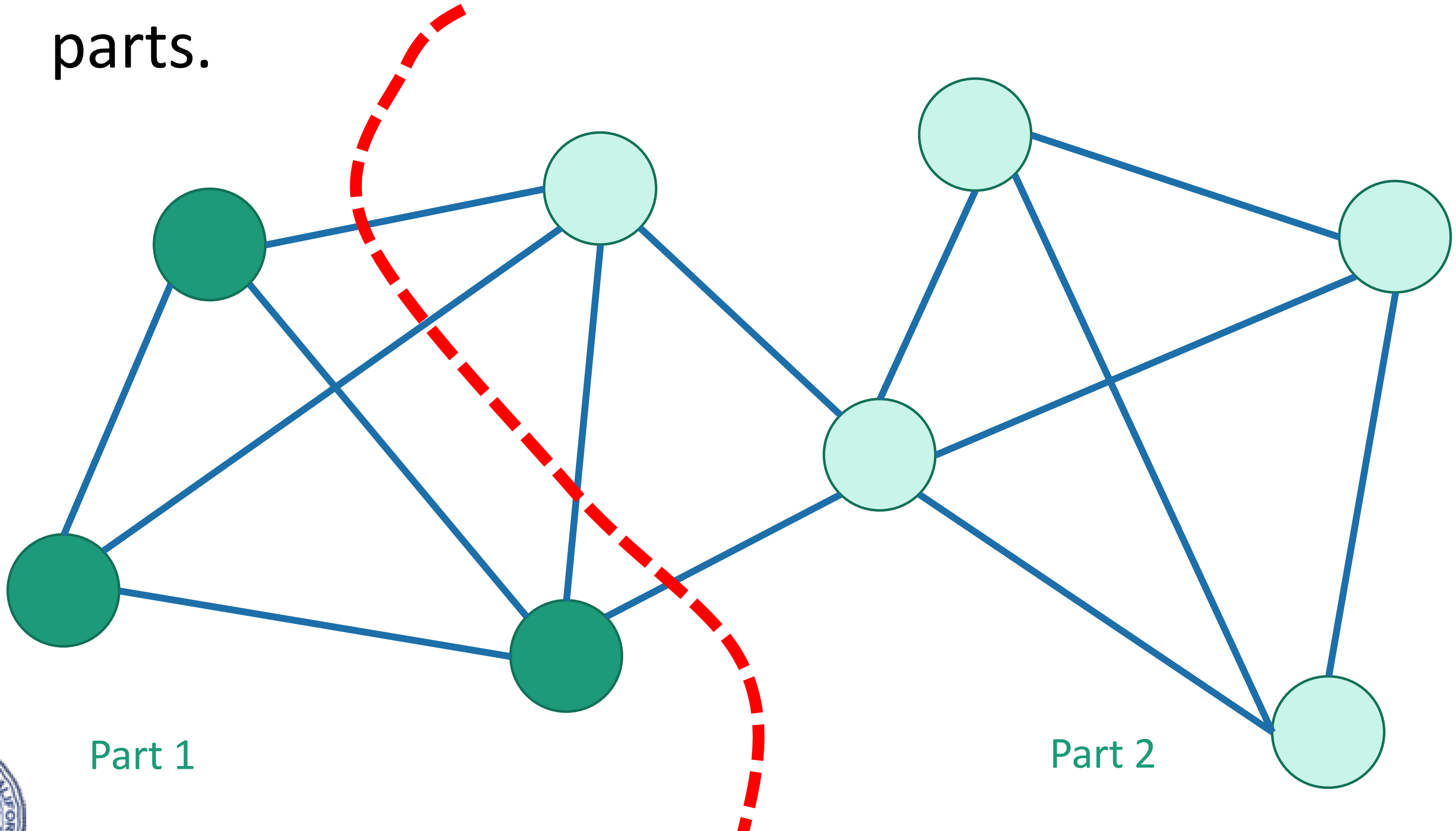
- A cut is a partition of the vertices into two nonempty parts.



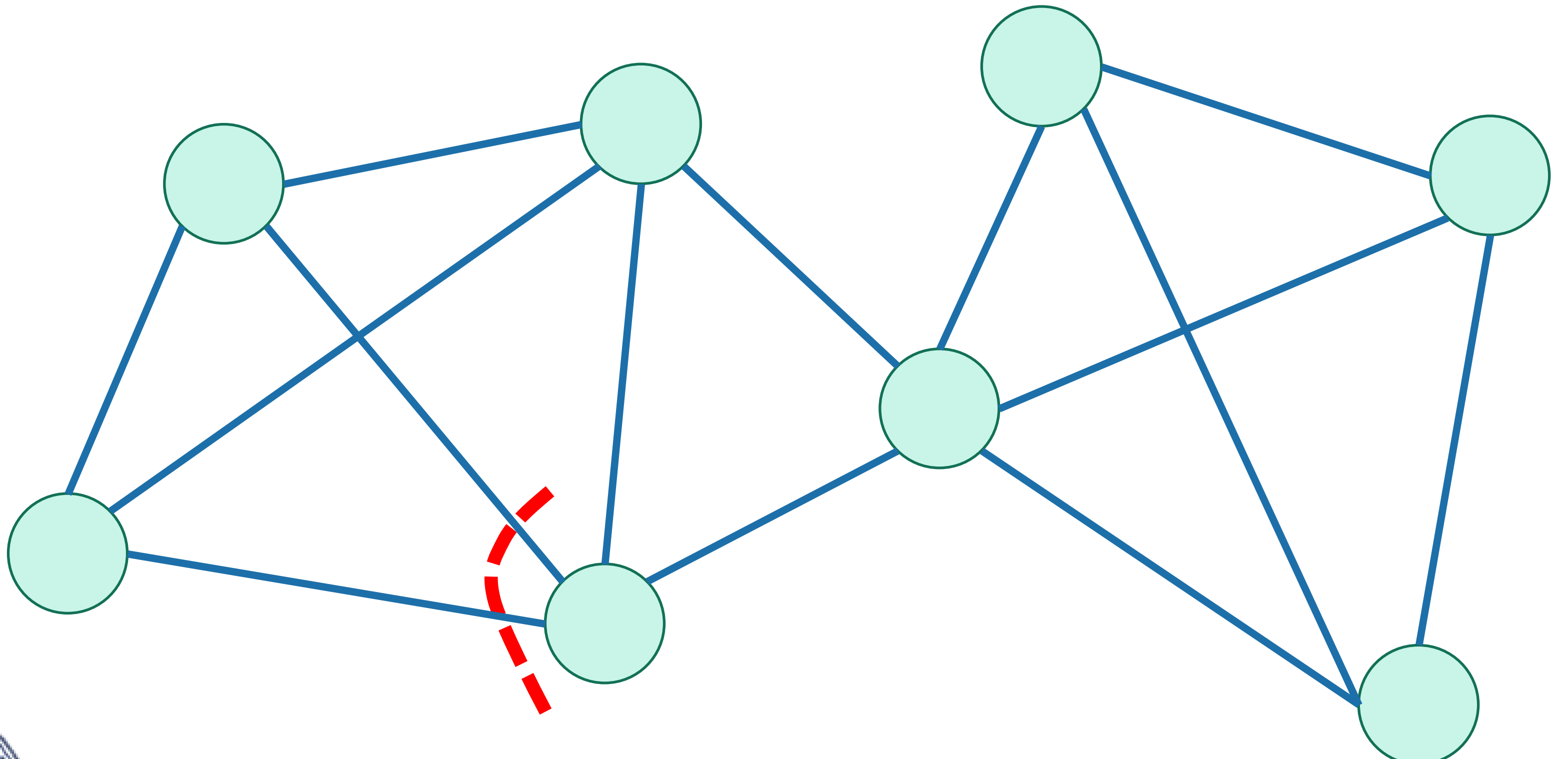
Recall: cuts in graphs

*For today, all graphs are **undirected and unweighted.**

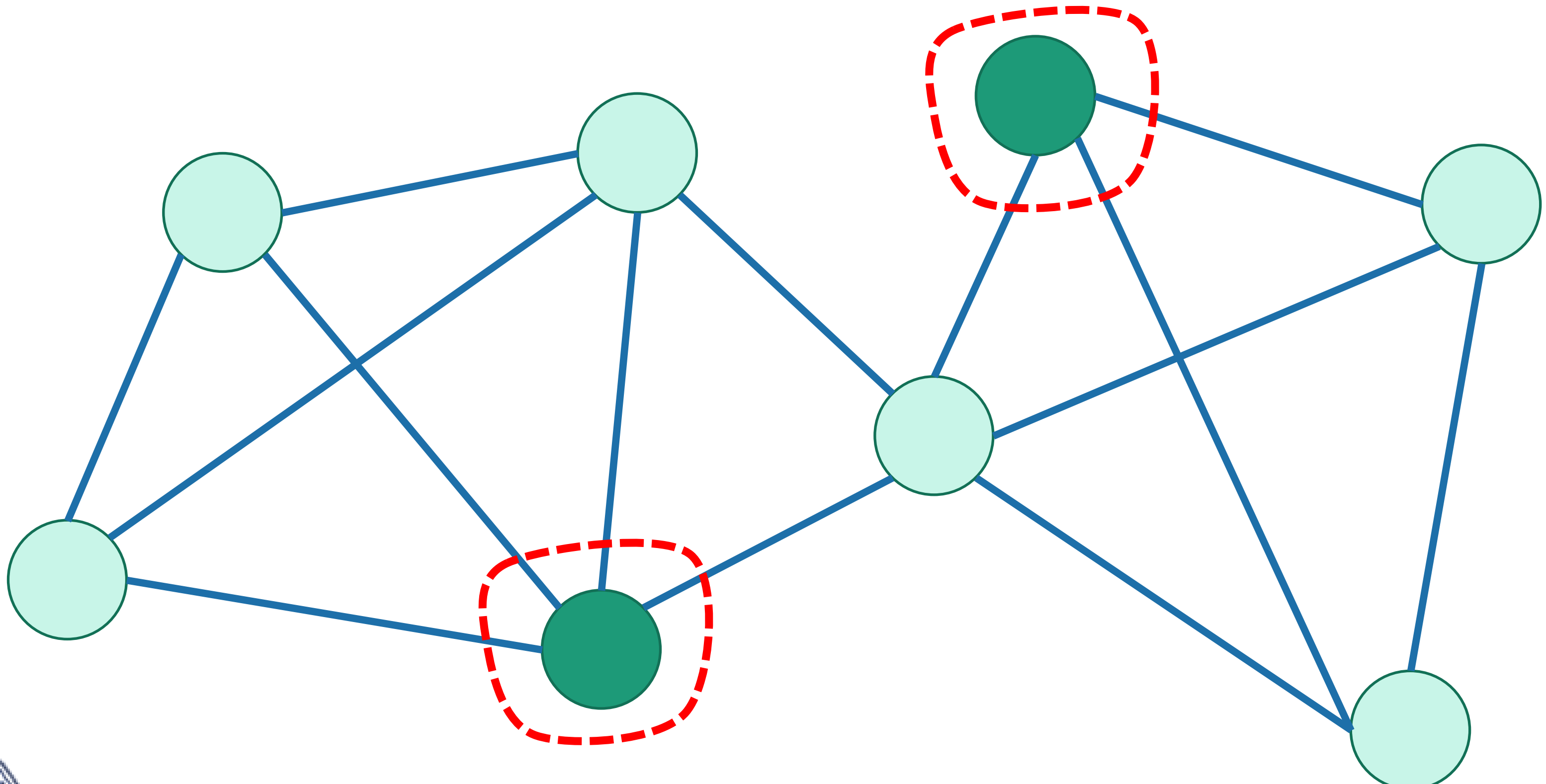
- A cut is a partition of the vertices into two nonempty parts.



This is not a cut



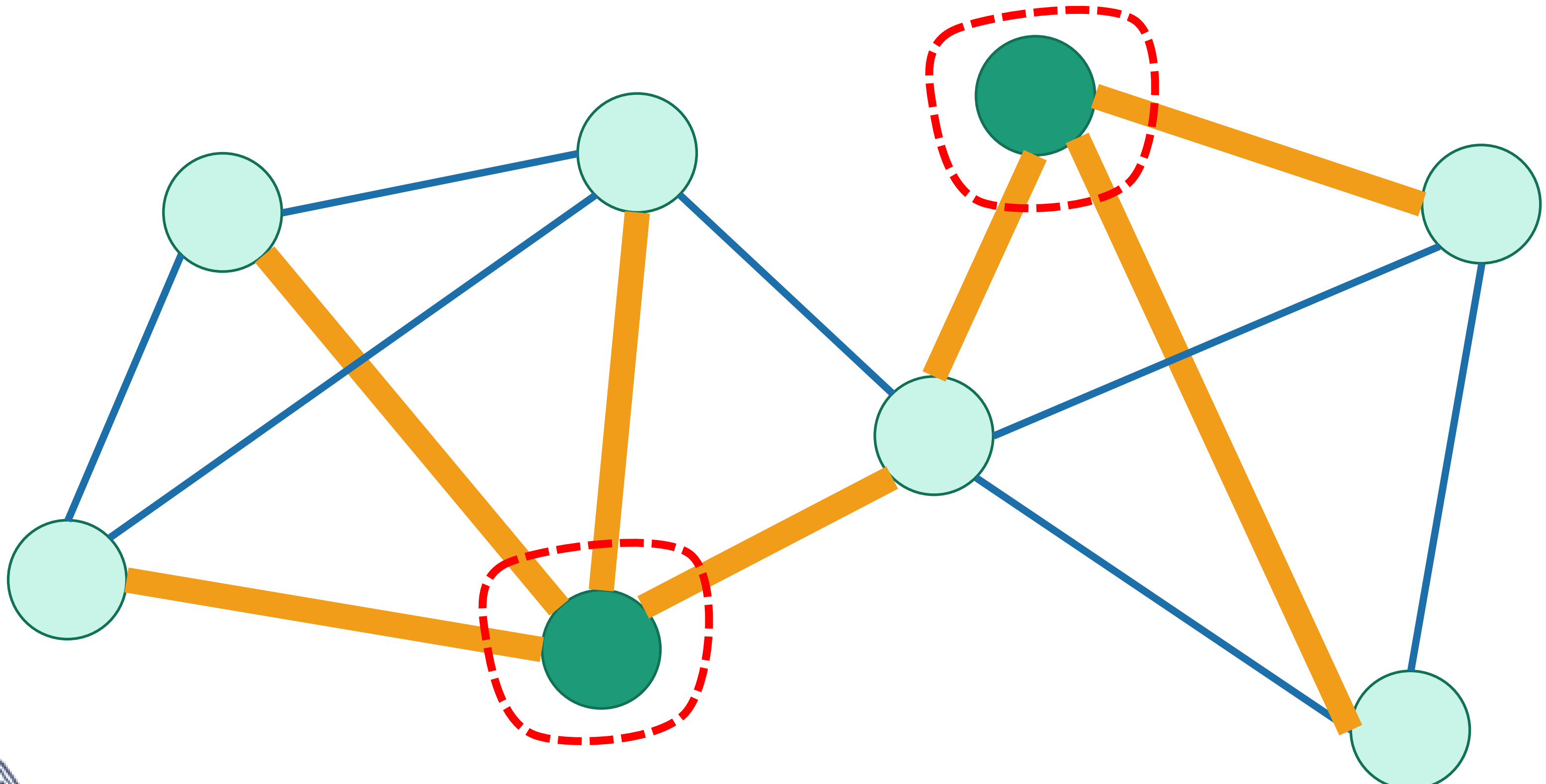
This is a cut



This is a cut

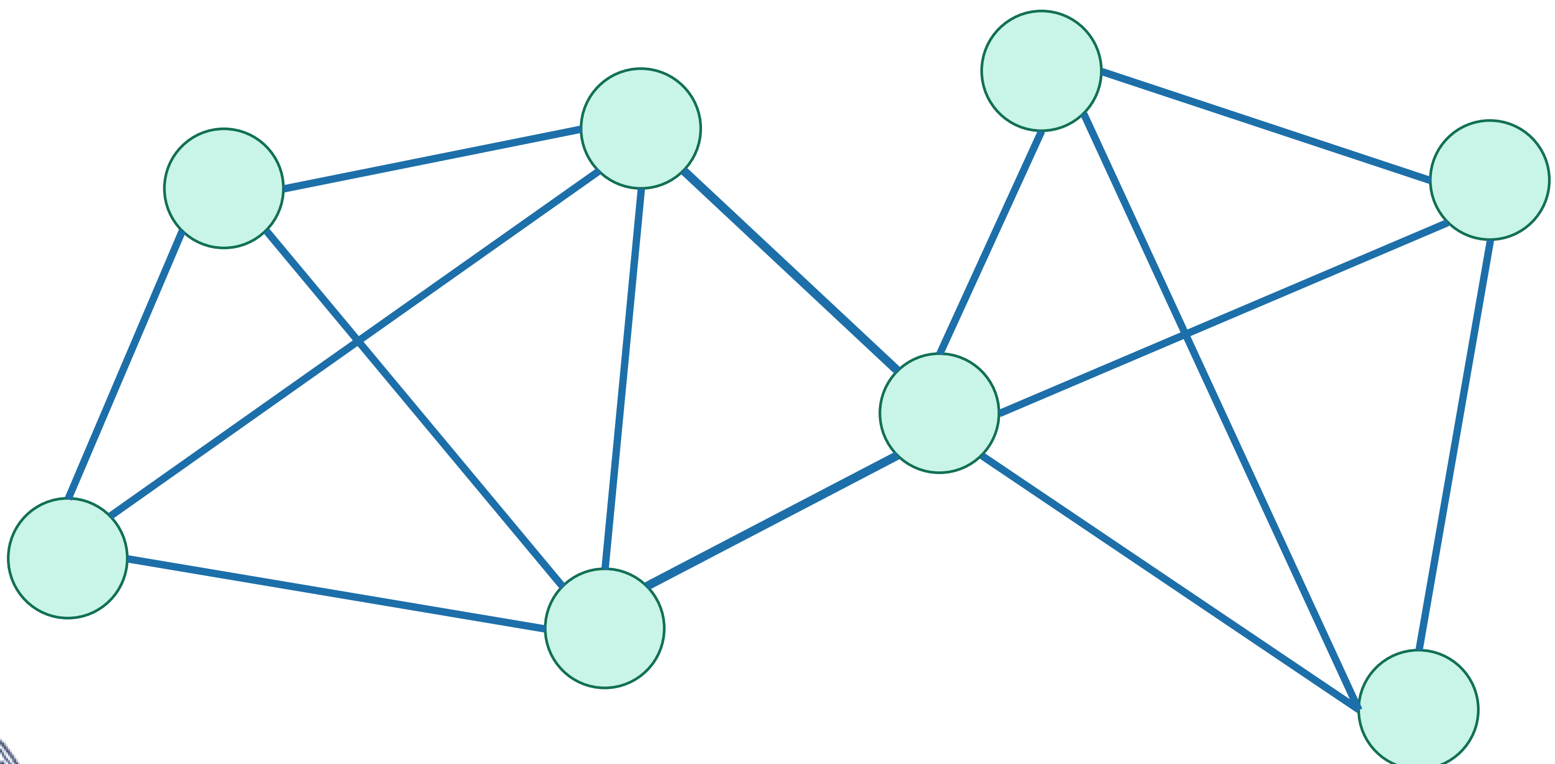
These edges **cross the cut.**

- They go from one part to the other.



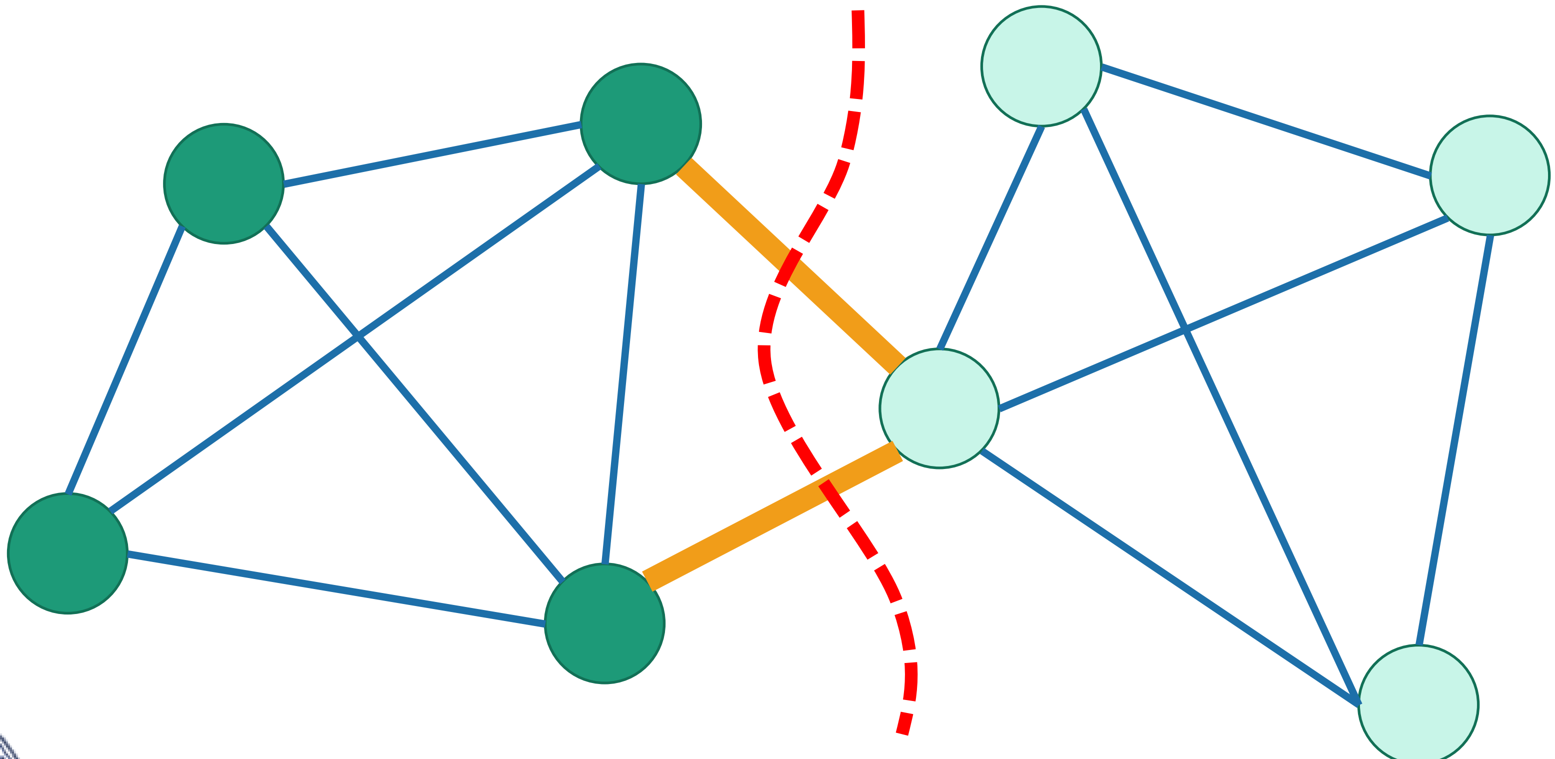
A (global) minimum cut

is a cut that has the fewest edges possible crossing it.



A (global) minimum cut

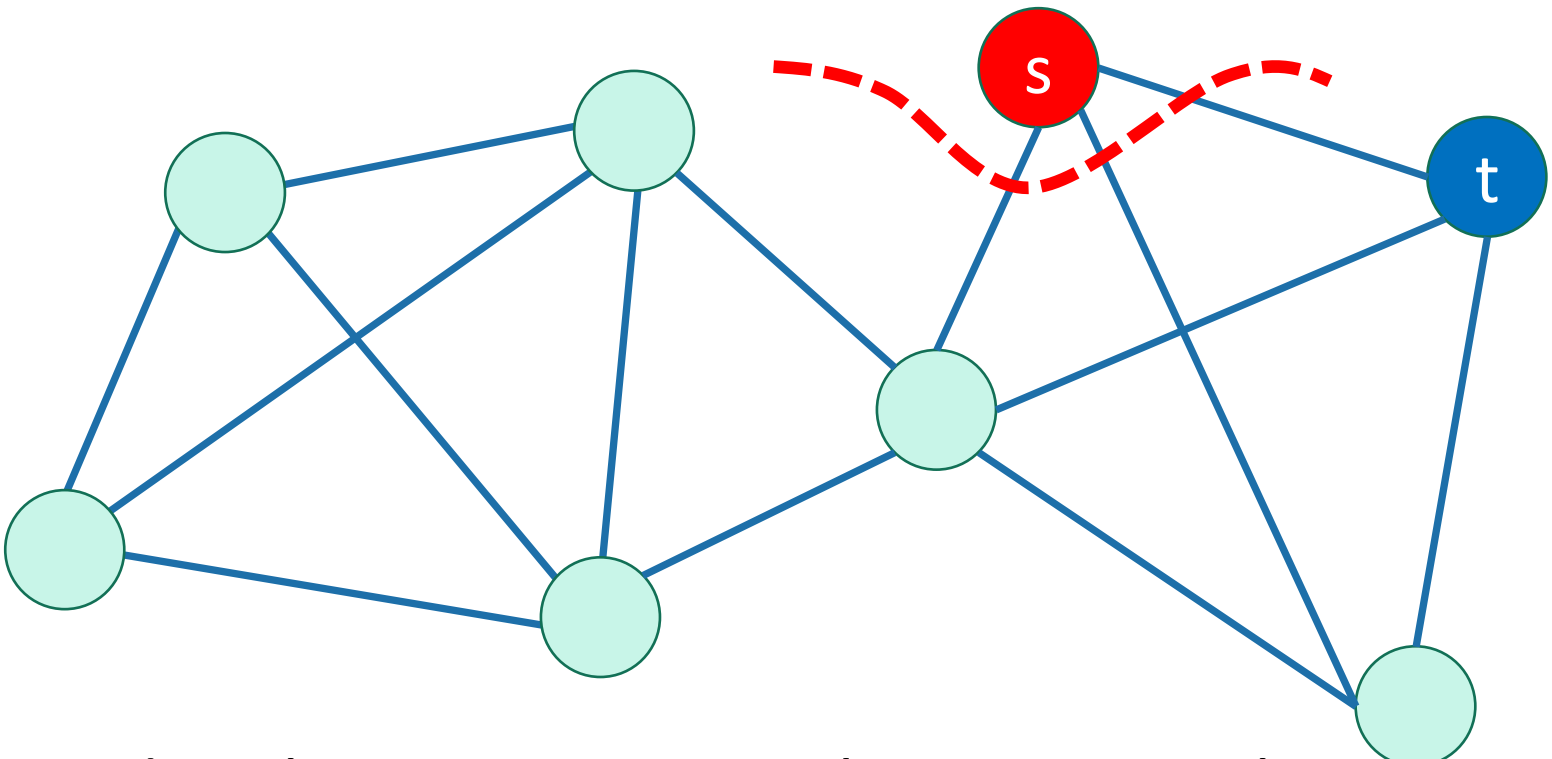
is a cut that has the fewest edges possible crossing it.



Why “global”?

- Next lecture we’ll talk about **min s-t cuts**

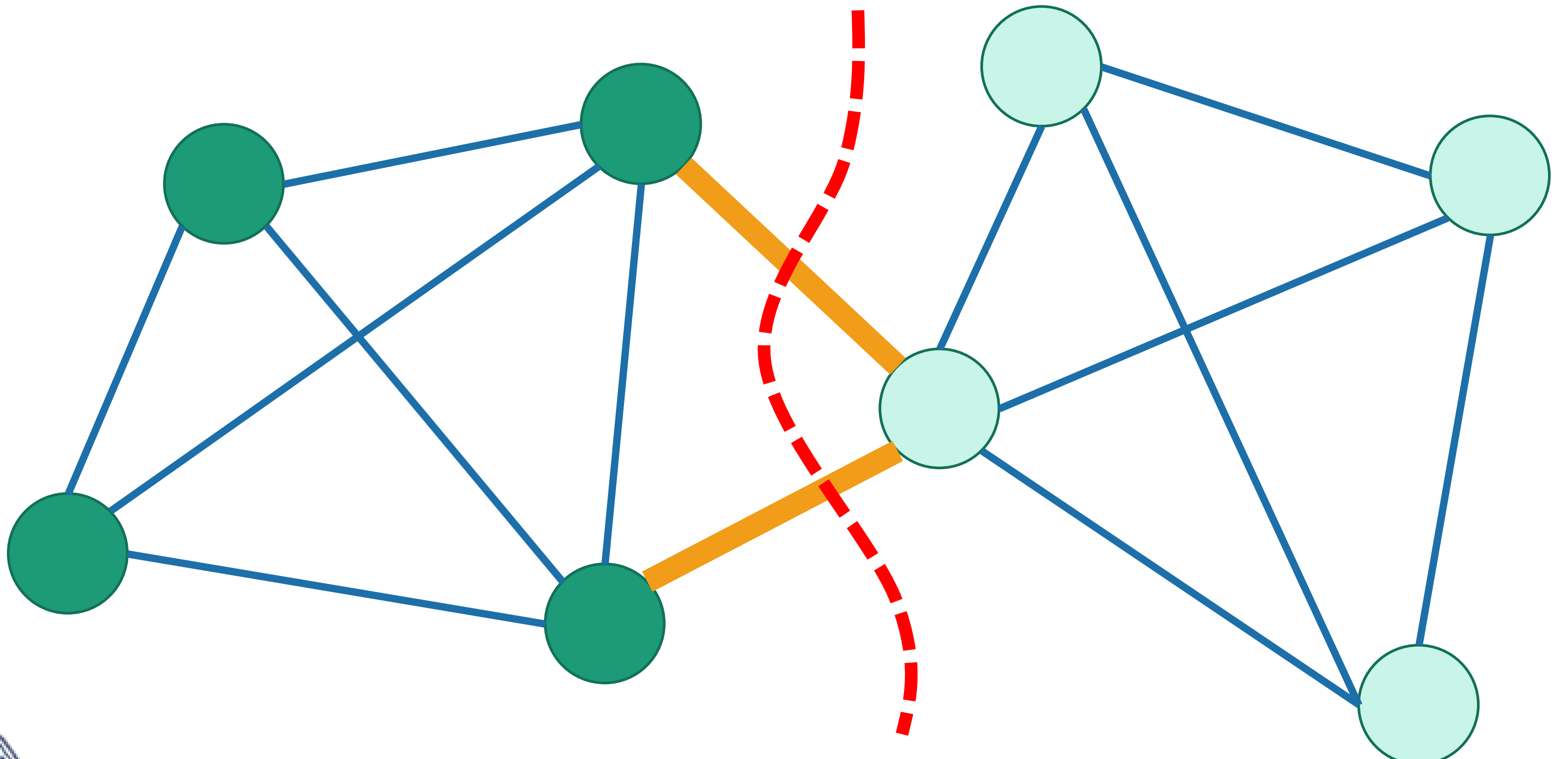
Minimum cut which separates a specified vertex s from t



- Today, there are no special vertices, so the minimum cut is “global.”

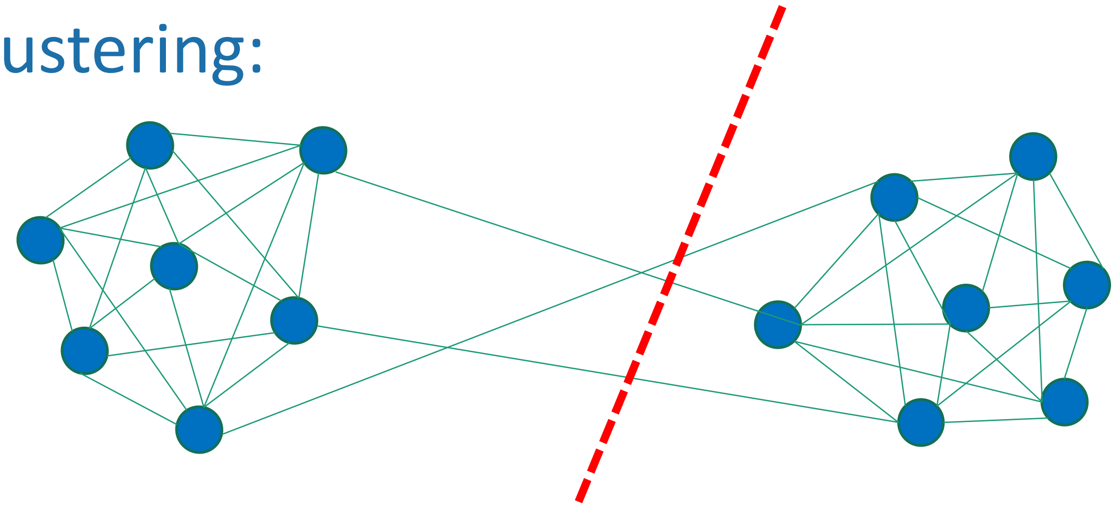
A (global) minimum cut

is a cut that has the fewest edges possible crossing it.

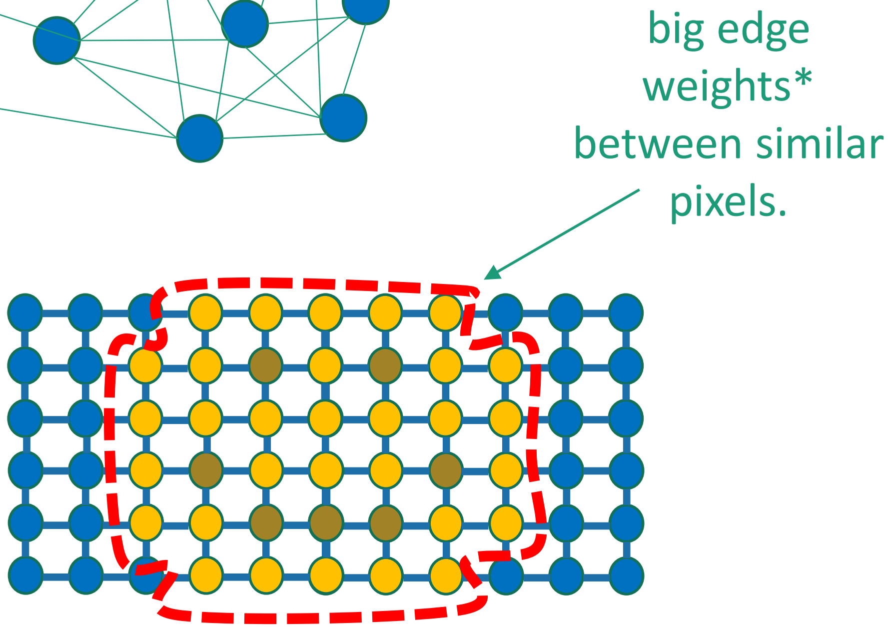


Why might we care about global minimum cuts?

- Clustering:



- Image Segmentation



*For the rest of today edges aren't weighted; but the algorithm can be adapted to deal with edge weights.

Karger's algorithm

- Finds **global minimum cuts** in undirected graphs
- Randomized algorithm
  - But a different sort of randomized algorithm than Quicksort!
- Karger's algorithm **might be wrong**.
  - While QuickSort, which just might be slow.
- Why would we want an algorithm that might be wrong?
  - **With high probability it won't be wrong.**
  - Maybe the stakes are low and the cost of a deterministic algorithm is high.

Different sorts of gambling

- QuickSort is a **Las Vegas randomized algorithm**
  - It is always correct.
  - It might be slow.

Yes, this is a technical term.

Formally:

- For all inputs A, QuickSort (A) returns a sorted array.
- For all inputs A, with high probability over the choice of pivots, QuickSort(A) runs quickly.

Different sorts of gambling

- Karger's Algorithm is a **Monte Carlo randomized algorithm**
  - It is always fast.
  - It might be wrong.

Formally:

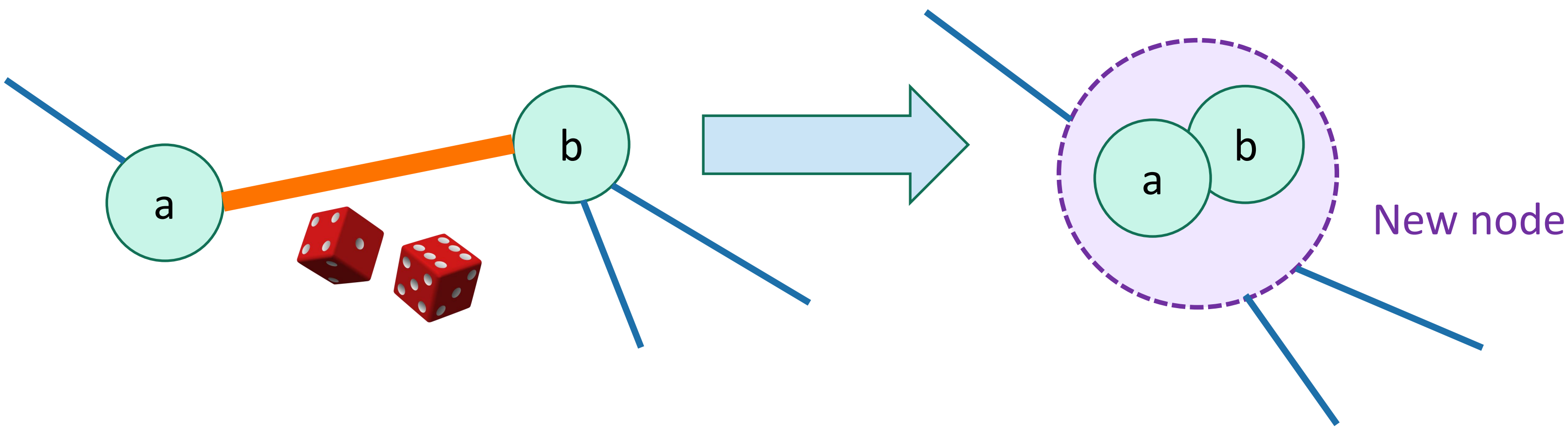
- For all inputs G, with probability at least ___ over the randomness in Karger's algorithm, Karger(G) returns a minimum cut.

- For all inputs G, with probability 1 Karger's algorithm runs in time no more than ___.

Algorithms that might be slow and might also be wrong are called "Atlantic City" algorithms

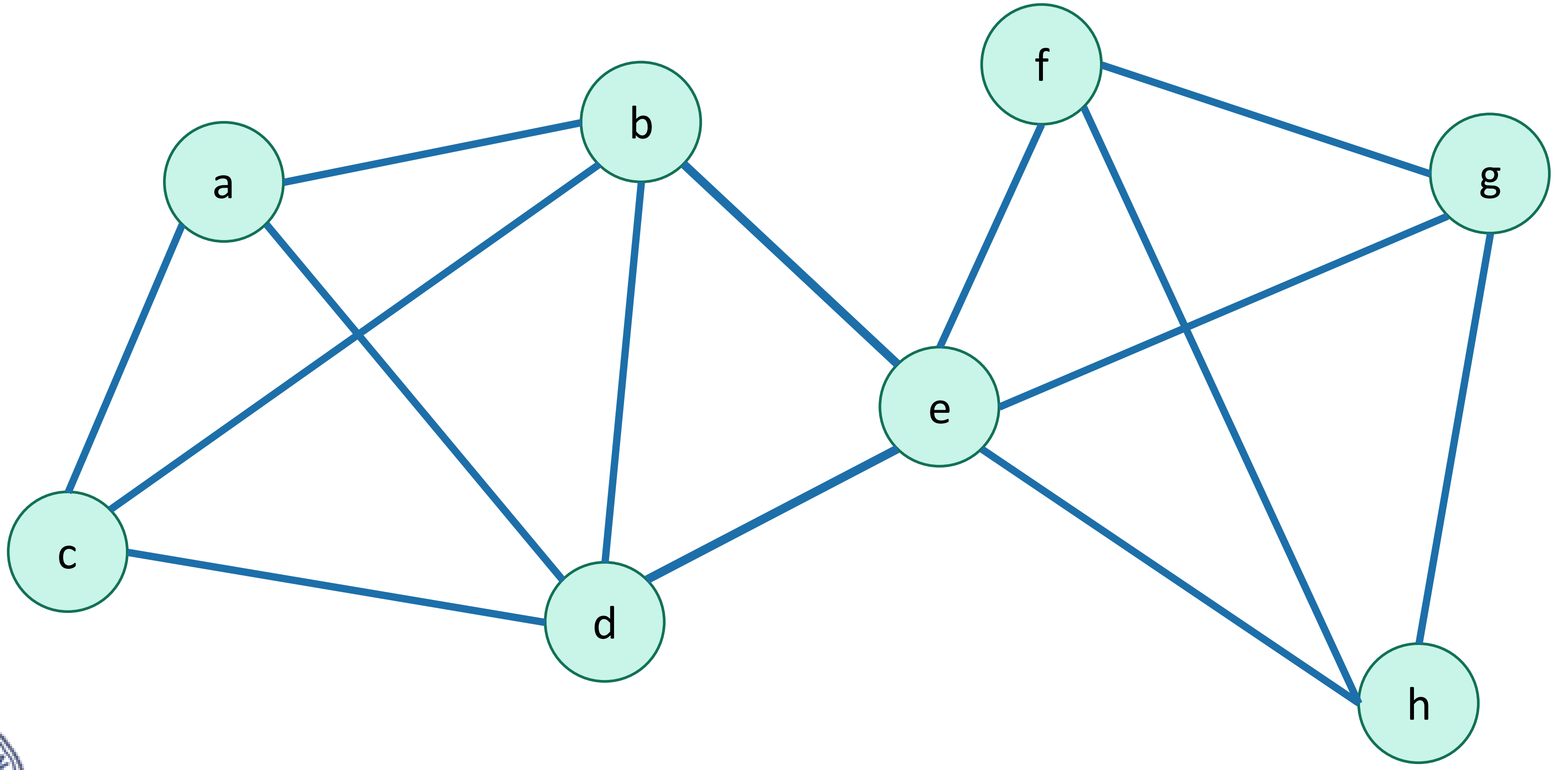
Karger's algorithm

- Pick a random edge.
- **Contract** it.
- Repeat until you only have two vertices left.

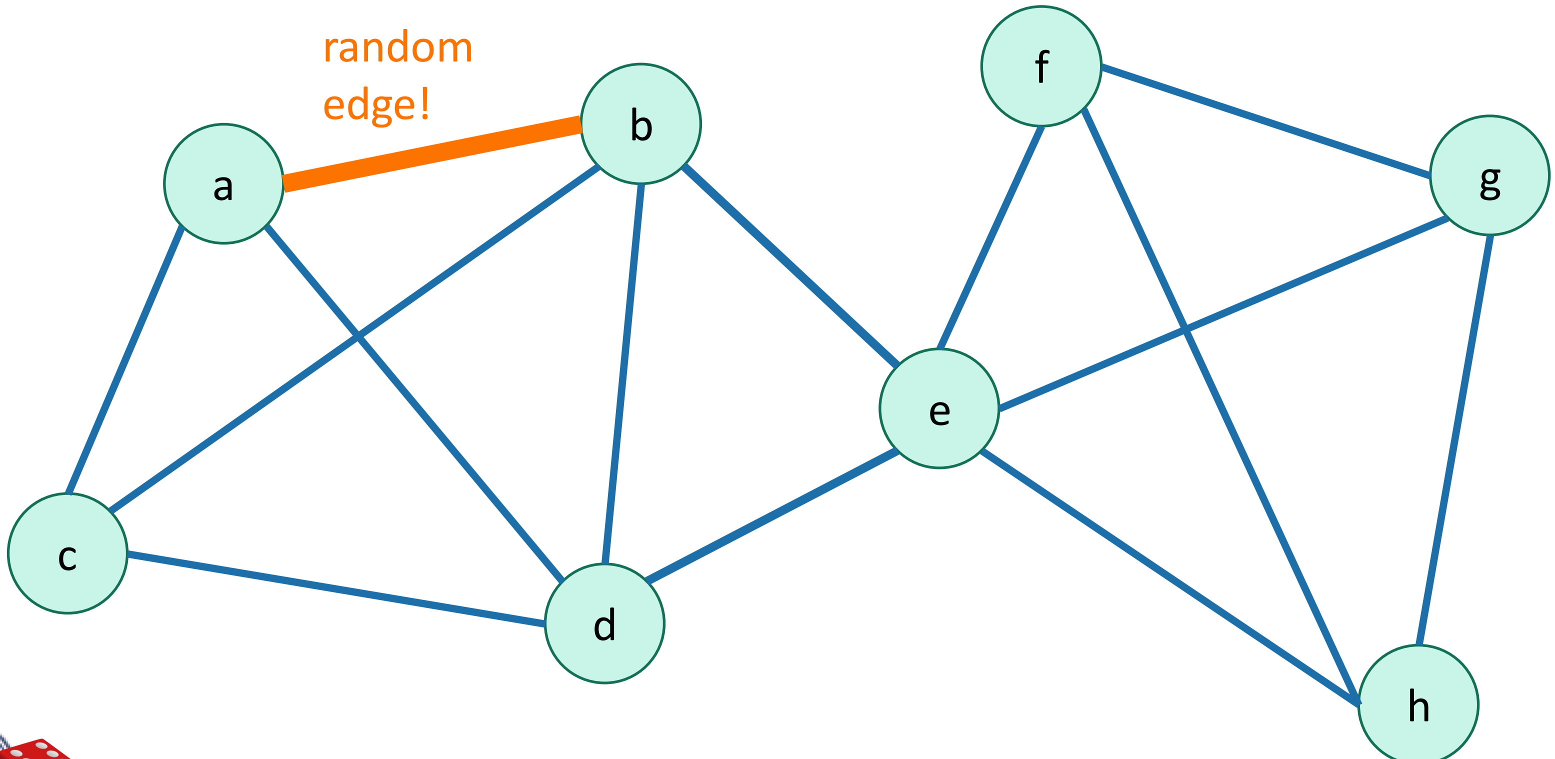


Why is this a good idea? We'll see shortly.

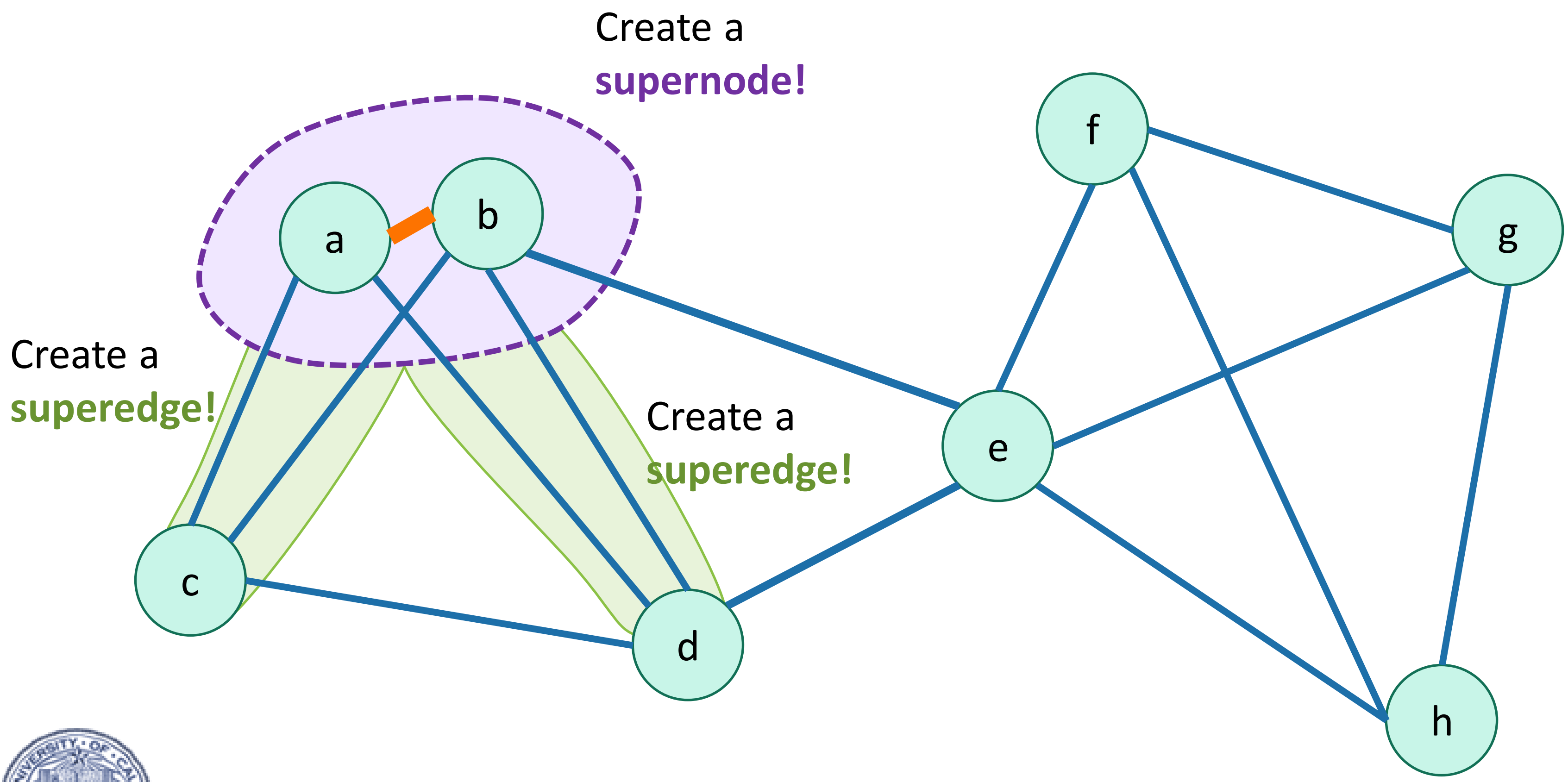
Karger’s algorithm



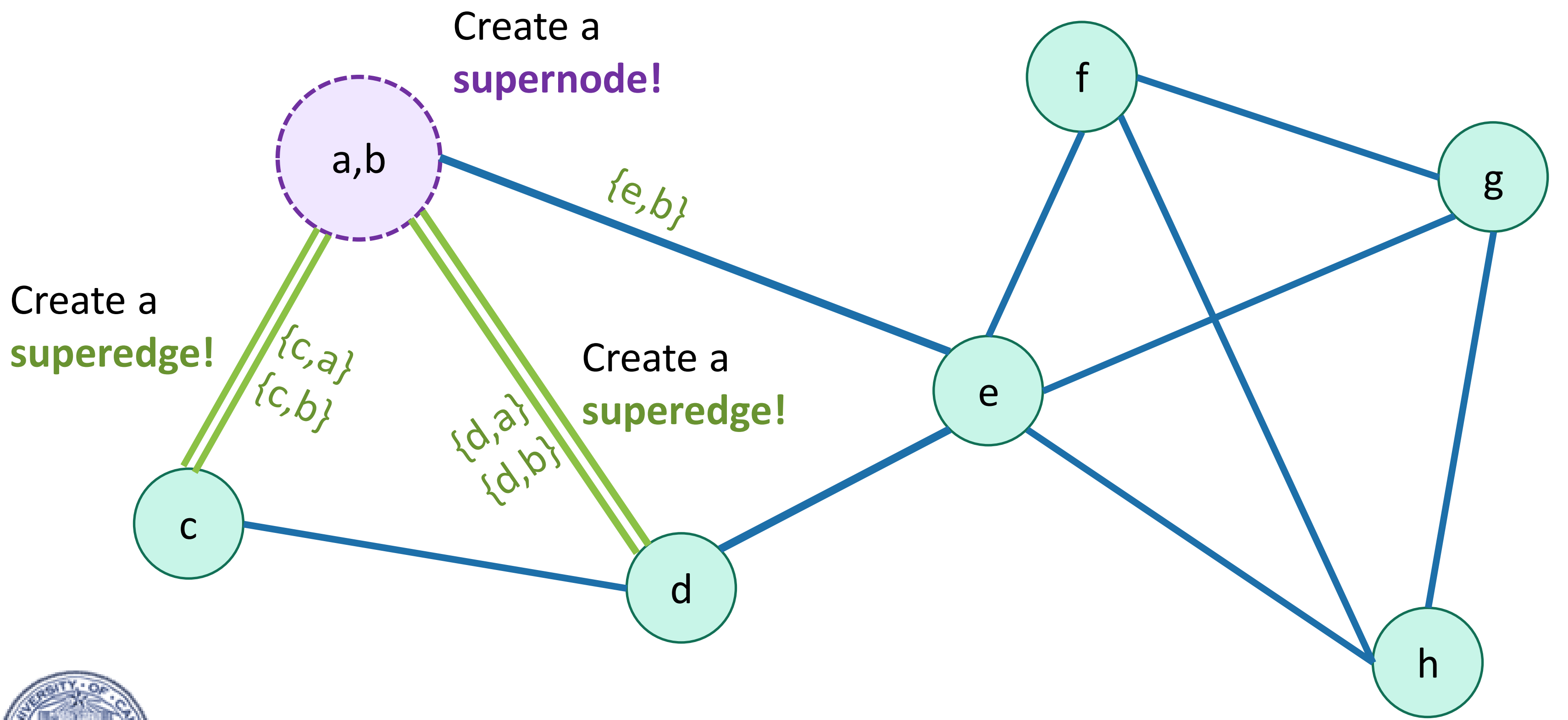
Karger’s algorithm



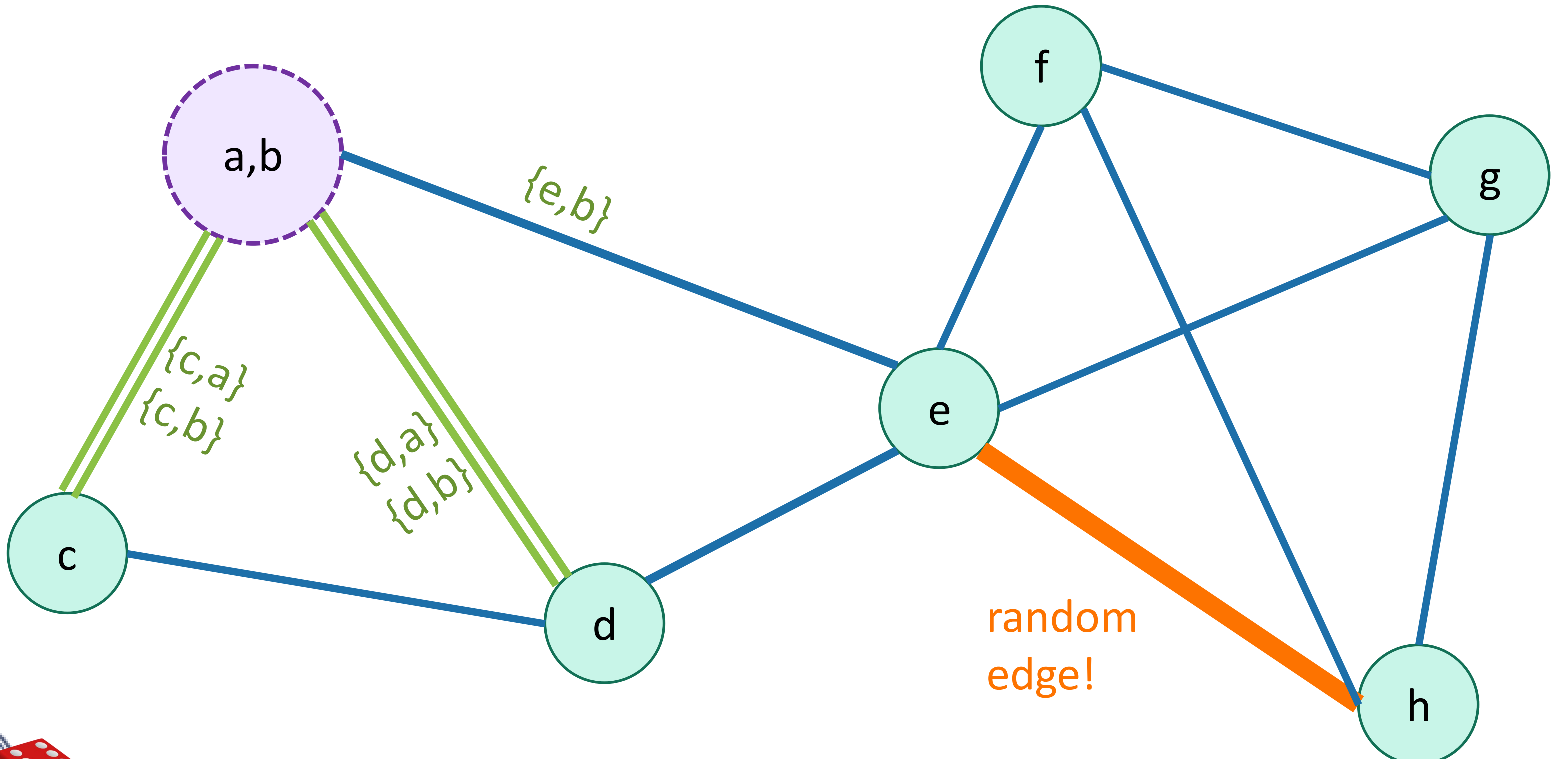
Karger’s algorithm



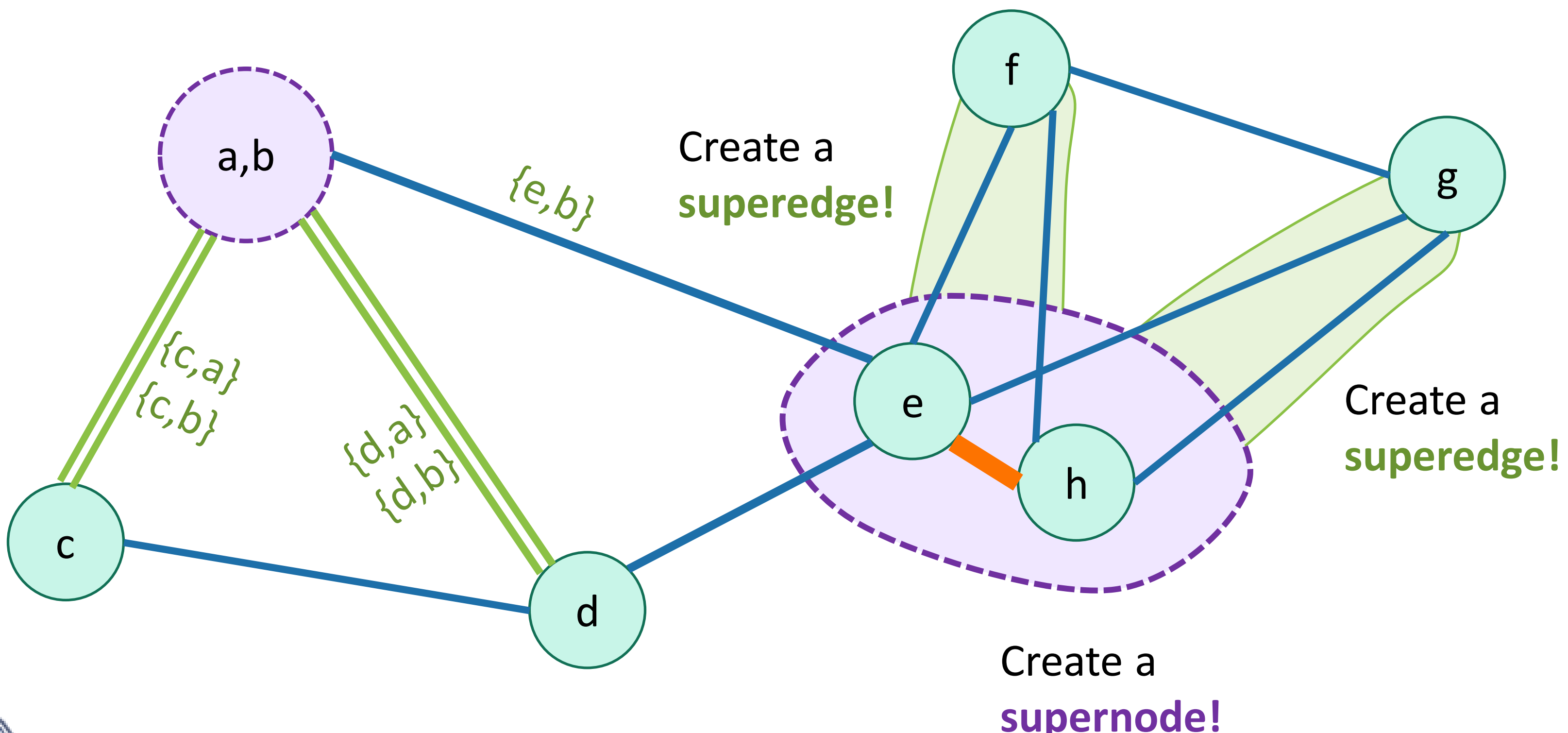
Karger's algorithm



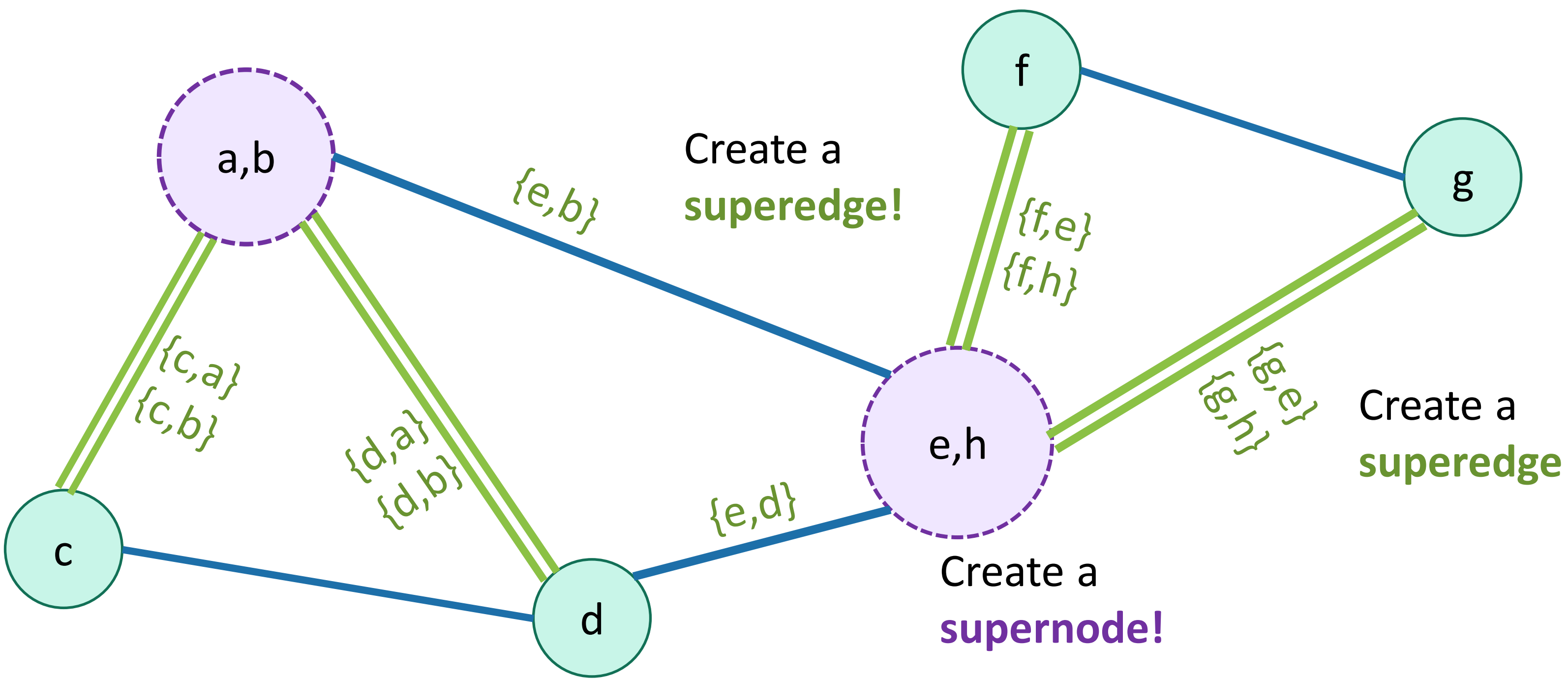
Karger's algorithm



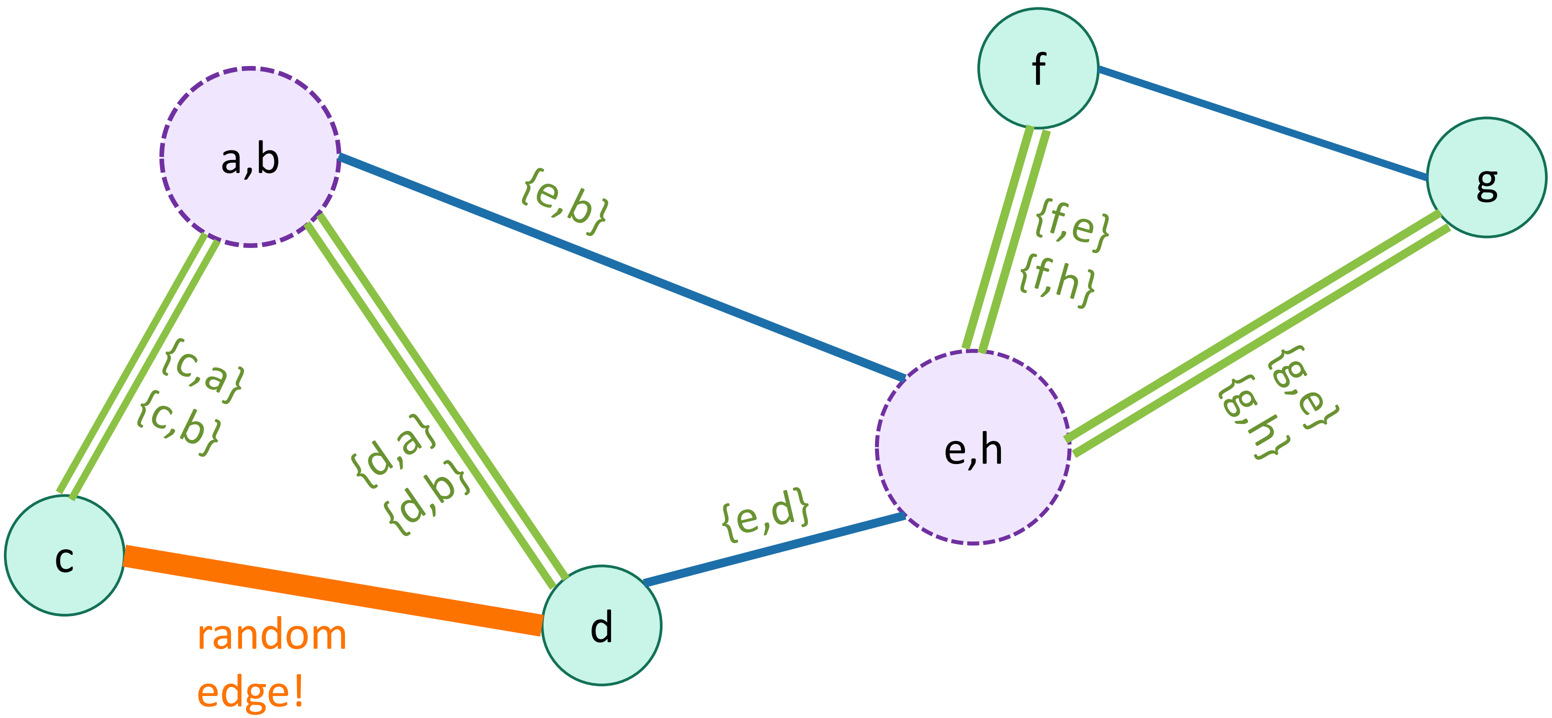
Karger's algorithm



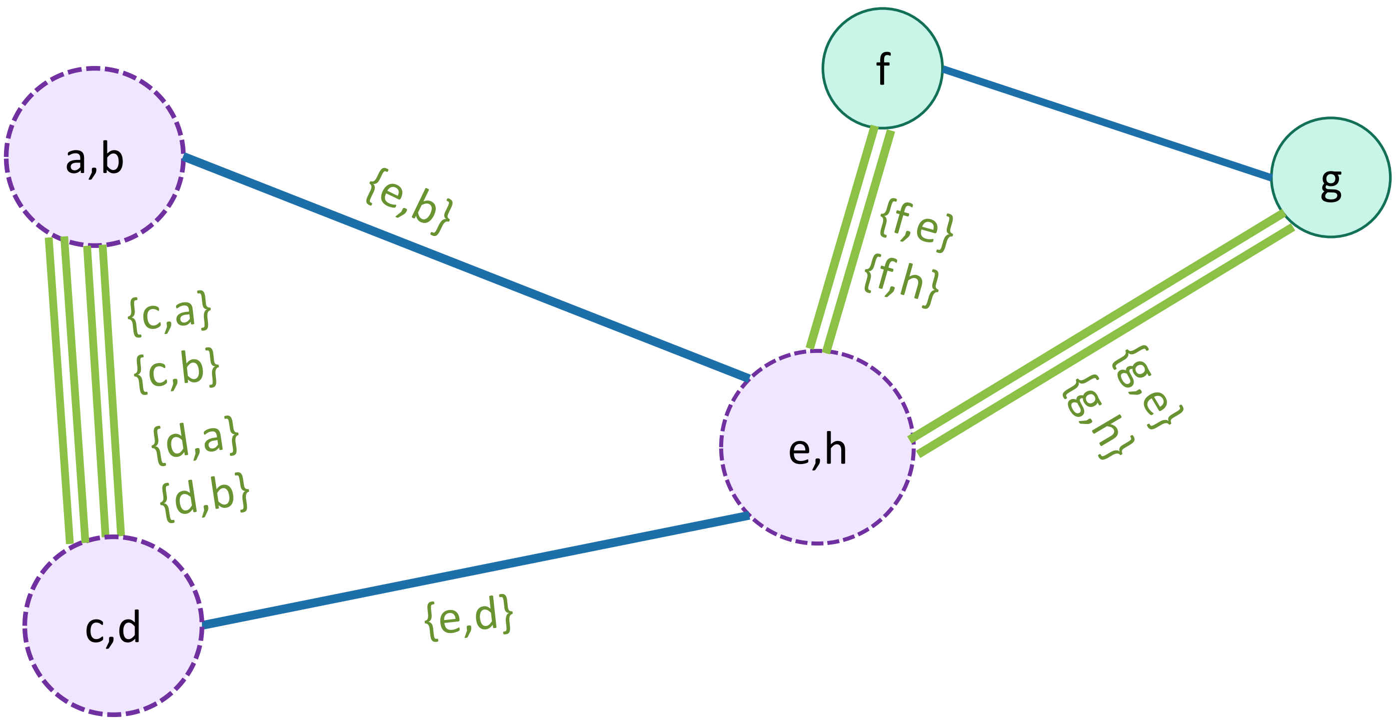
Karger’s algorithm



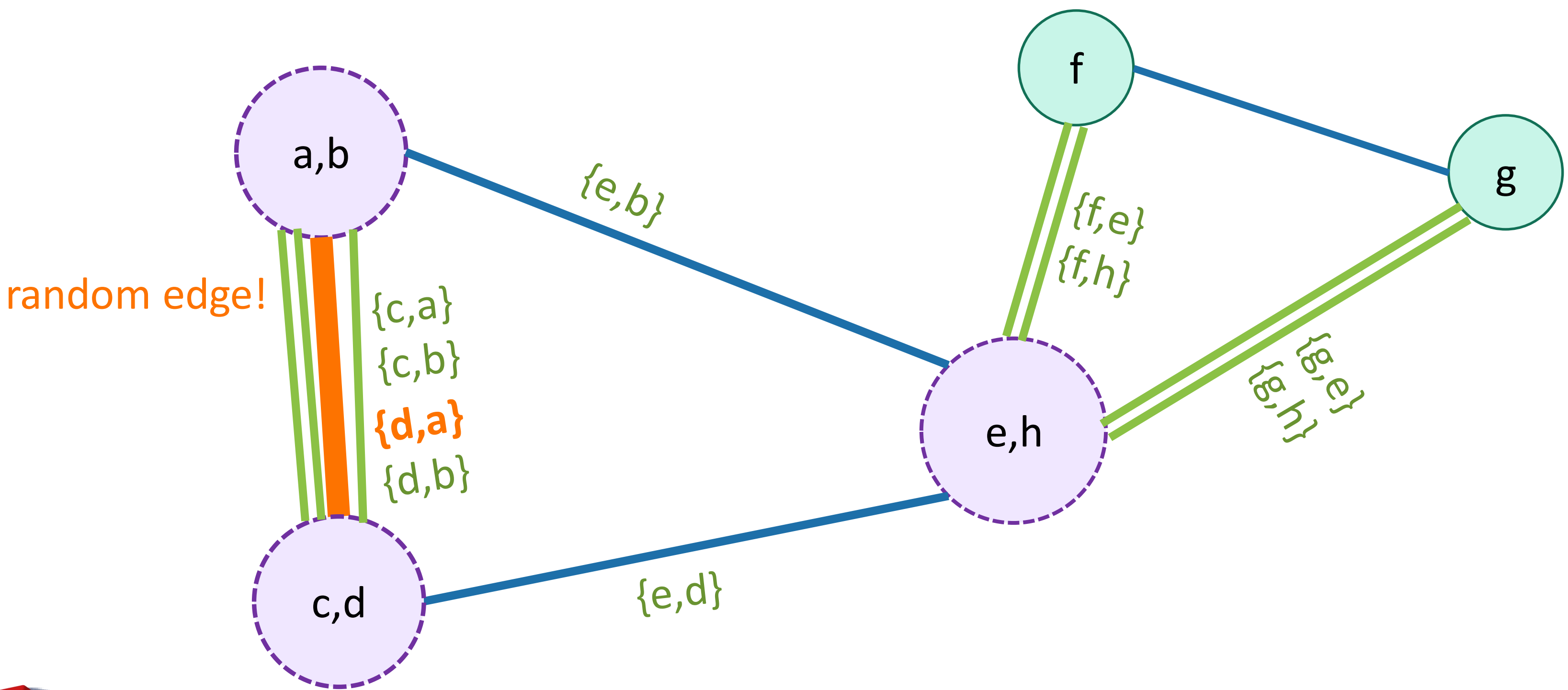
Karger's algorithm



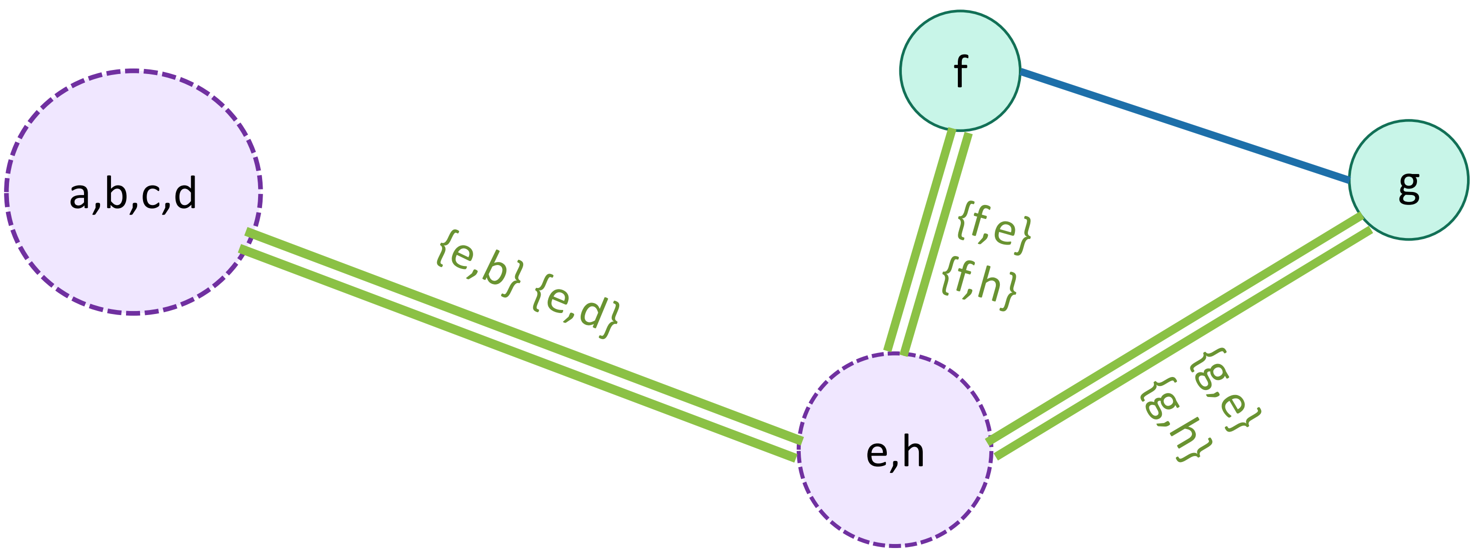
Karger's algorithm



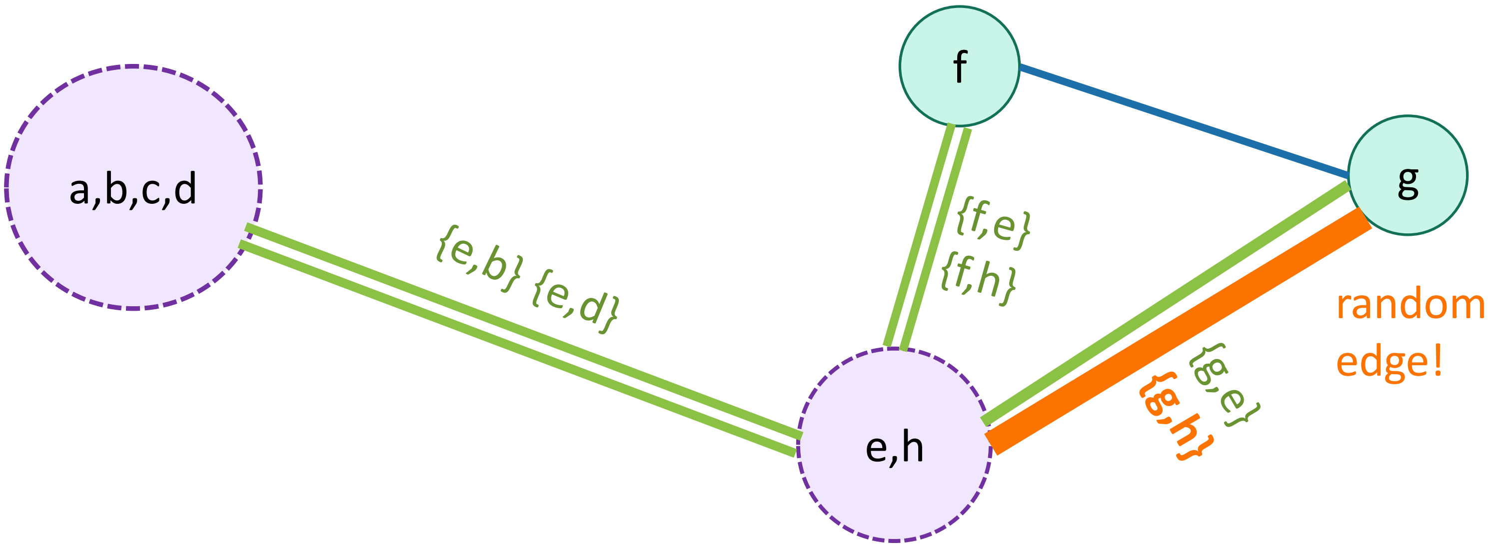
Karger's algorithm



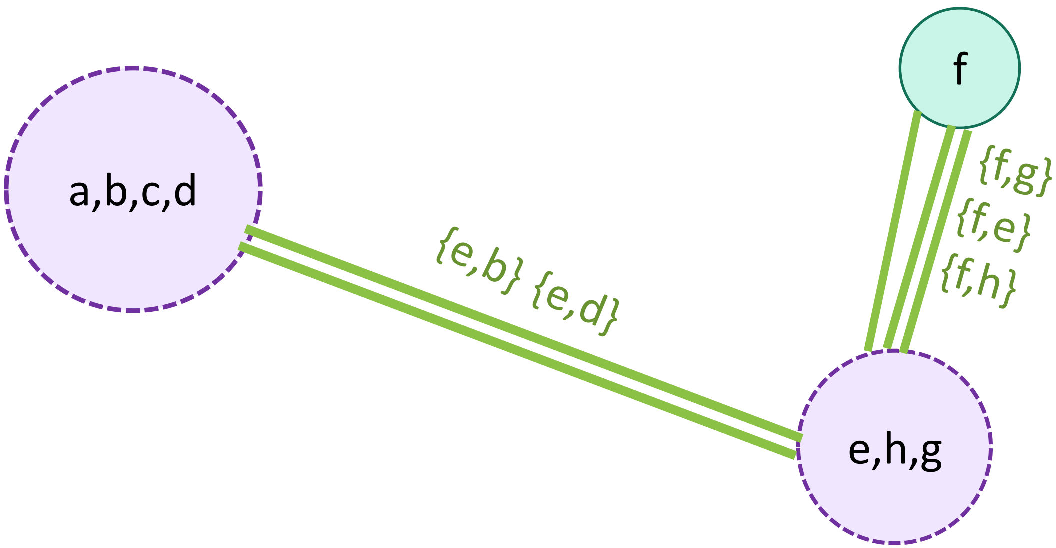
Karger's algorithm



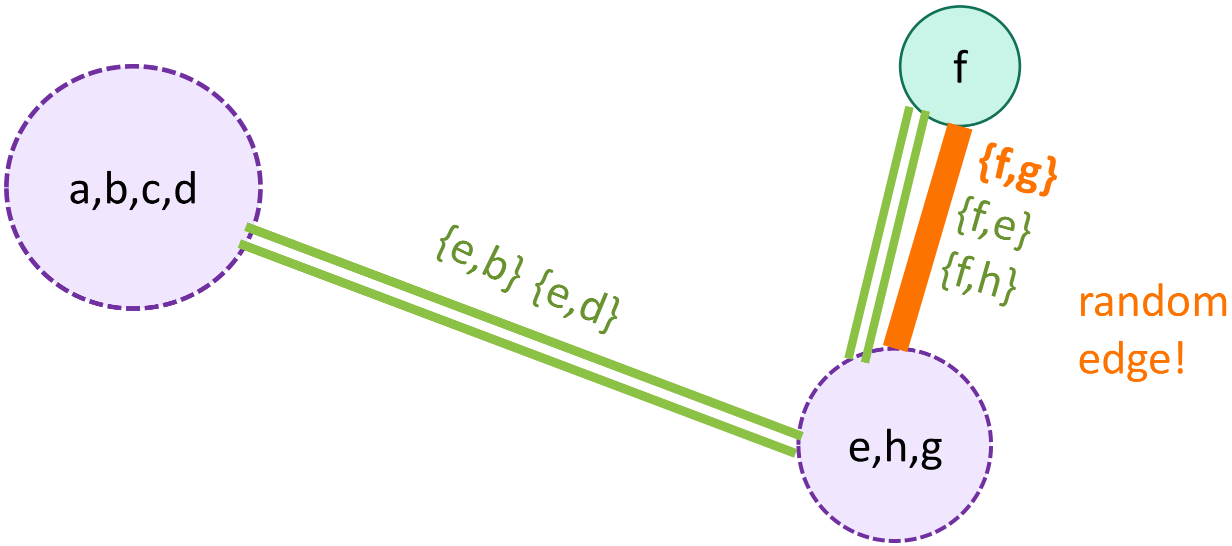
Karger's algorithm



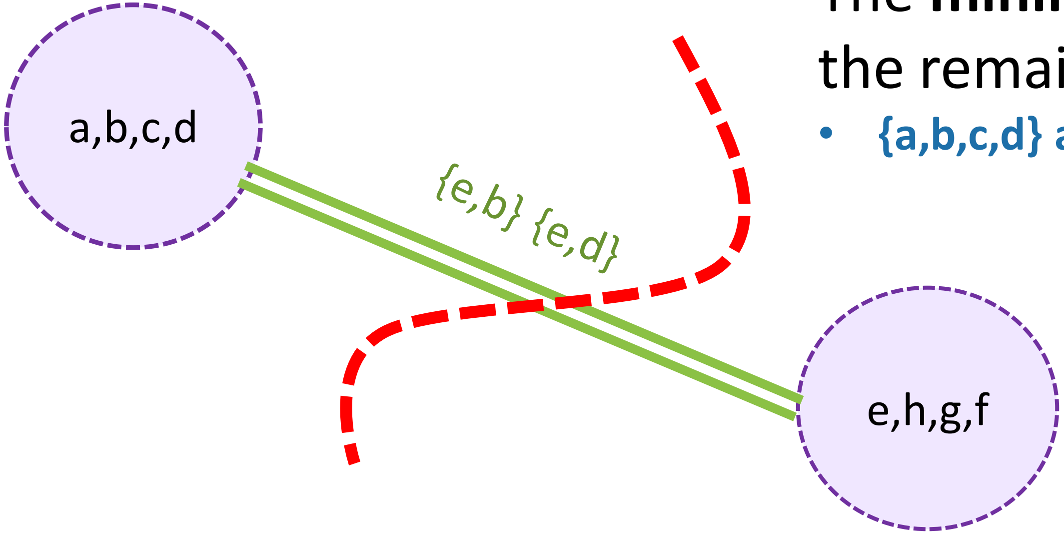
Karger's algorithm



Karger's algorithm



Karger's algorithm



Now stop!

- There are only two nodes left.

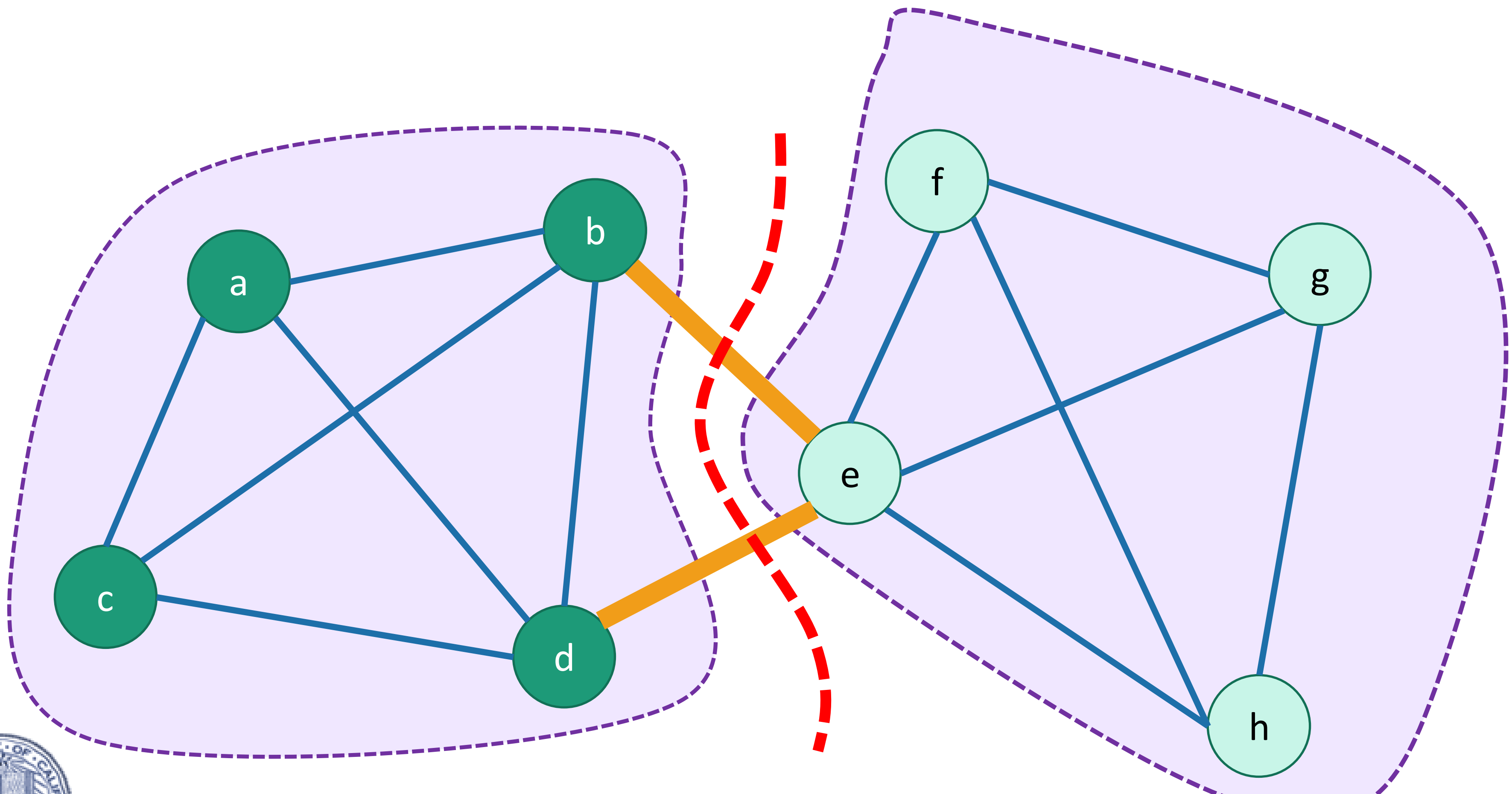
The **minimum cut** is given by the remaining super-nodes:

- **{a,b,c,d} and {e,h,f,g}**

Karger's algorithm

The **minimum cut** is given by the remaining super-nodes:

- **{a,b,c,d} and {e,h,f,g}**



Karger’s algorithm

- Does it work?

- Is it fast?

How do we implement this?

- See next slide with pseudocode
  - This maintains a secondary "superGraph" which keeps track of superNodes and superEdges
- Running time?
  - We contract n-2 edges
    - Each time we contract an edge we get rid of a vertex, and we get rid of n – 2 vertices total.
  - Naively each contraction takes time O(n)
    - Maybe there are $\Omega(n)$ nodes in the superNodes that we are merging. (We can do better with fancy data structures).
  - So total running time O(n^2).
    - We can do $O(m \cdot \alpha(n))$ with a union-find data structure, but $O(n^2)$ is good enough for today.

Pseudocode

Let $\overline{\boldsymbol{u}}$ denote the SuperNode in Γ containing u
Say $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ is the SuperEdge between $\overline{\boldsymbol{u}}, \overline{\boldsymbol{v}}$.

- **Karger**(G=(V,E)):
  - Γ = { SuperNode(v): v in V } // one supernode for each vertex
  - $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ = {(u,v)} for (u,v) in E // one superedge for each edge
  - $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ = {} for (u,v) not in E.
  - F = copy of E // we'll choose randomly from F
  - **while** $|\Gamma| > 2$:
    - (u,v) ← uniformly random edge in F
    - **merge**(u, v)

      // merge the SuperNode containing u with the SuperNode containing v.
    - $F \leftarrow F \setminus E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$

      // remove all the edges in the SuperEdge between those SuperNodes.
  - **return** the cut given by the remaining two superNodes.

The **while** loop runs n-2 times

merge takes time O(n) naively

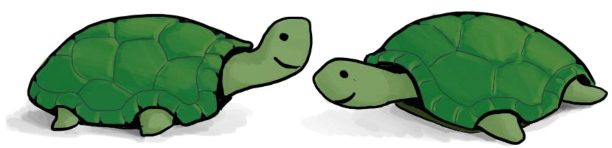
- **merge**(u, v): // merge also knows about Γ and the $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$'s
  - $\overline{\boldsymbol{x}}$ = SuperNode($\overline{\boldsymbol{u}} \cup \overline{\boldsymbol{v}}$) // create a new supernode
  - for each $\overline{\boldsymbol{w}}$ in $\Gamma \setminus \{\overline{\boldsymbol{u}}, \overline{\boldsymbol{v}}\}$:
    - $E_{\overline{\boldsymbol{x}},\overline{\boldsymbol{w}}} = E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{w}}} \cup E_{\overline{\boldsymbol{v}},\overline{\boldsymbol{w}}}$
  - Remove $\overline{\boldsymbol{u}}$ and $\overline{\boldsymbol{v}}$ from Γ and add $\overline{\boldsymbol{x}}$.

total runtime $O(n^2)$

We can do a bit better with fancy data structures, but let's go with this for now.

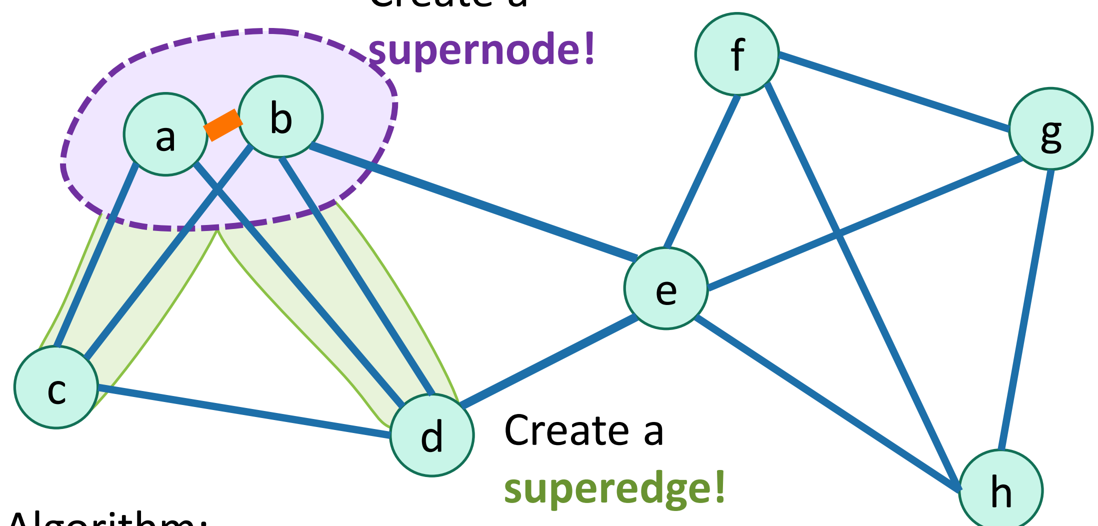
Karger’s algorithm

- Does it work?



Think-pair-share!

- Is it fast?
  - $O(n^2)$



Algorithm:

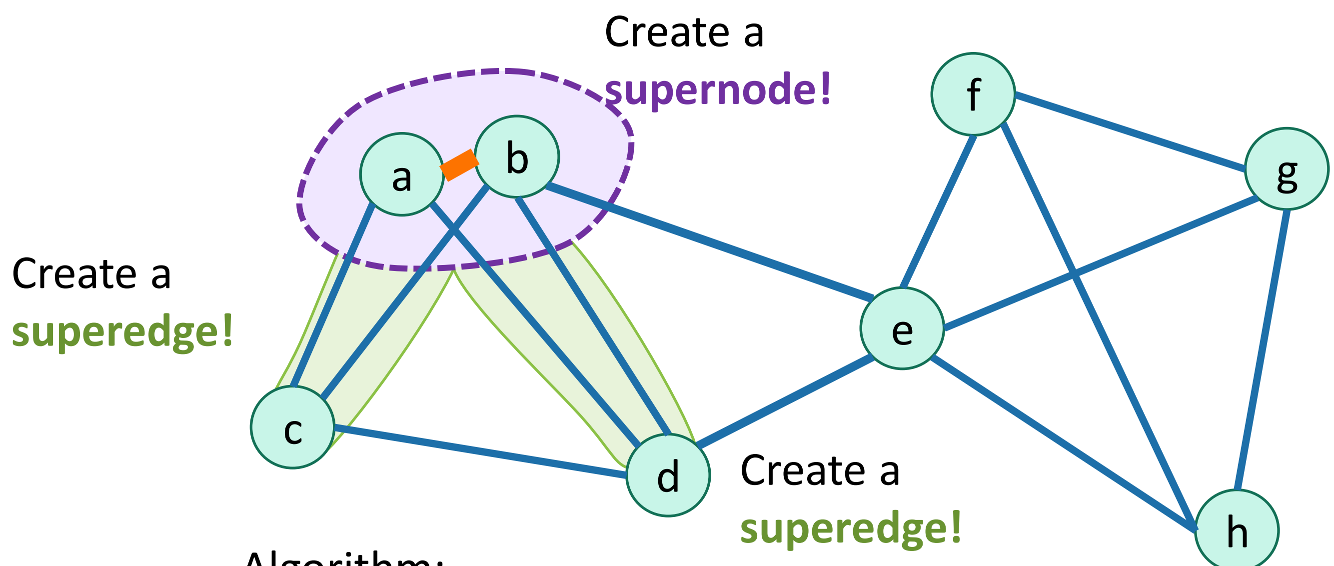
- Randomly contract edges until there are only two supernodes left.

Karger's algorithm

- Does it work?

  No?

- Is it fast?
  - $O(n^2)$

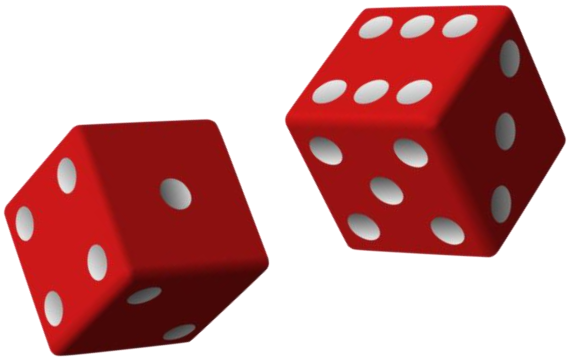


Algorithm:

- Randomly contract edges until there are only two supernodes left.

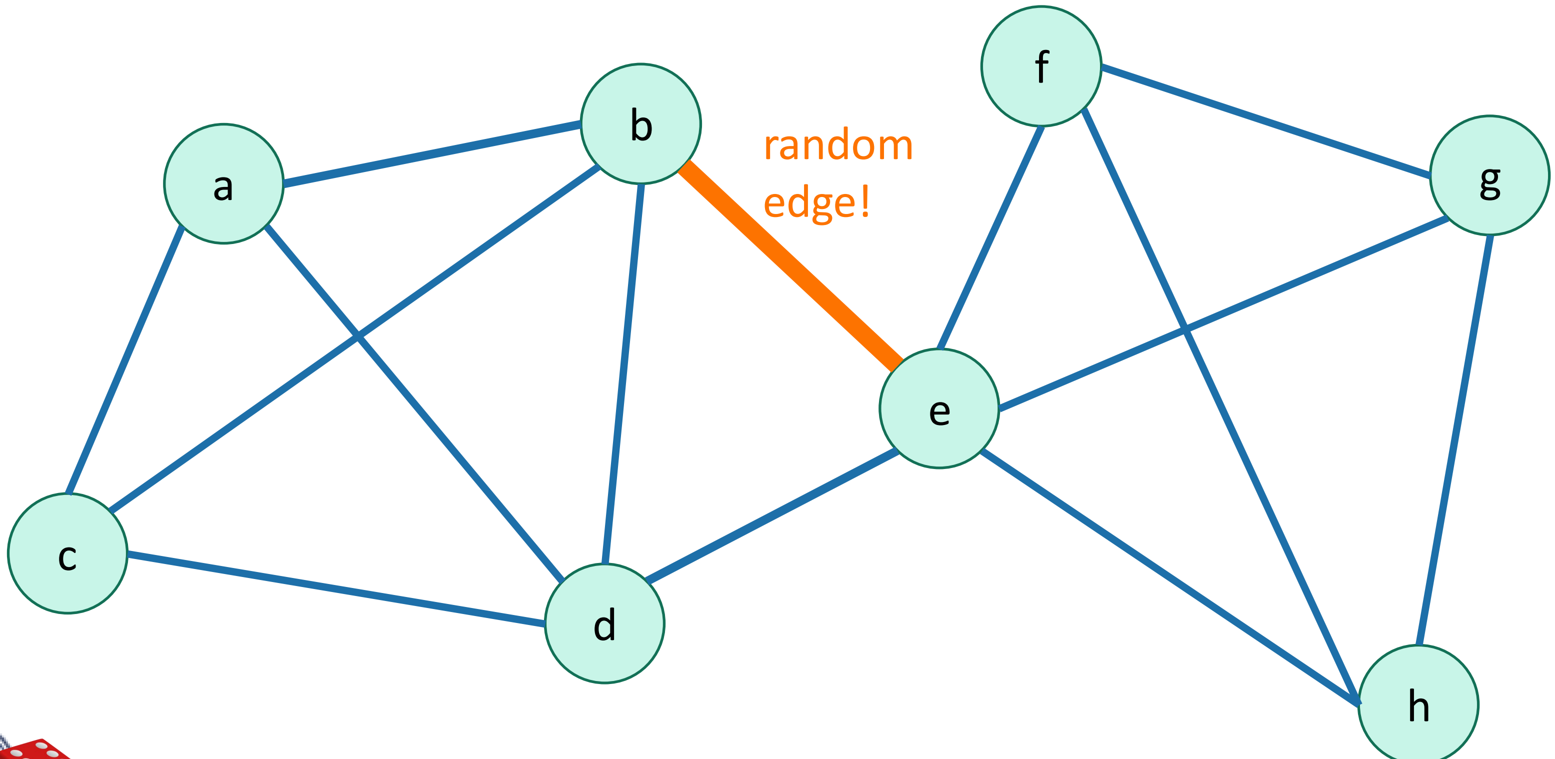
Why did that work?

- We got really lucky!
- This could have gone wrong in so many ways.



Karger's algorithm

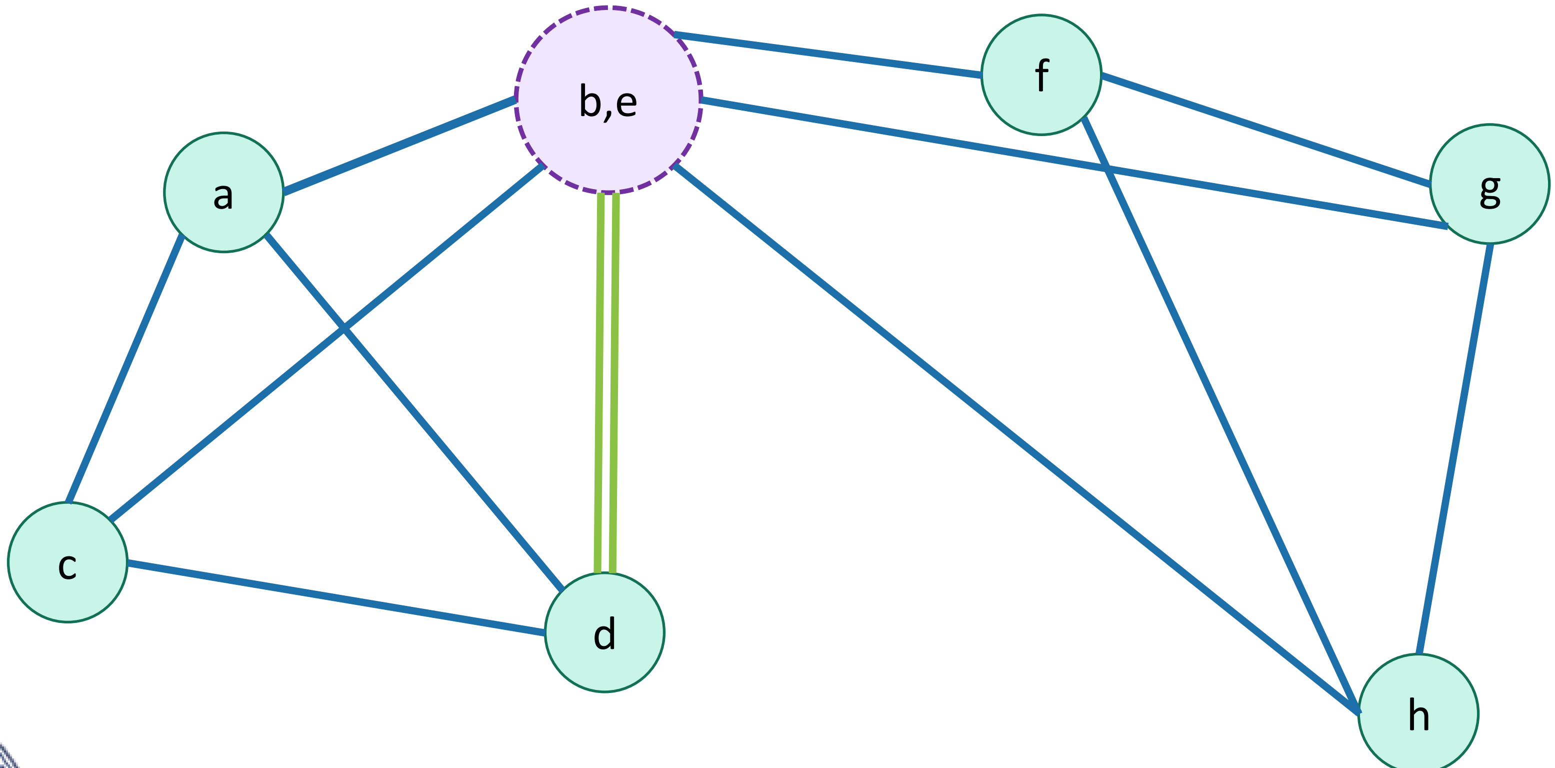
Say we had chosen this edge



Karger’s algorithm

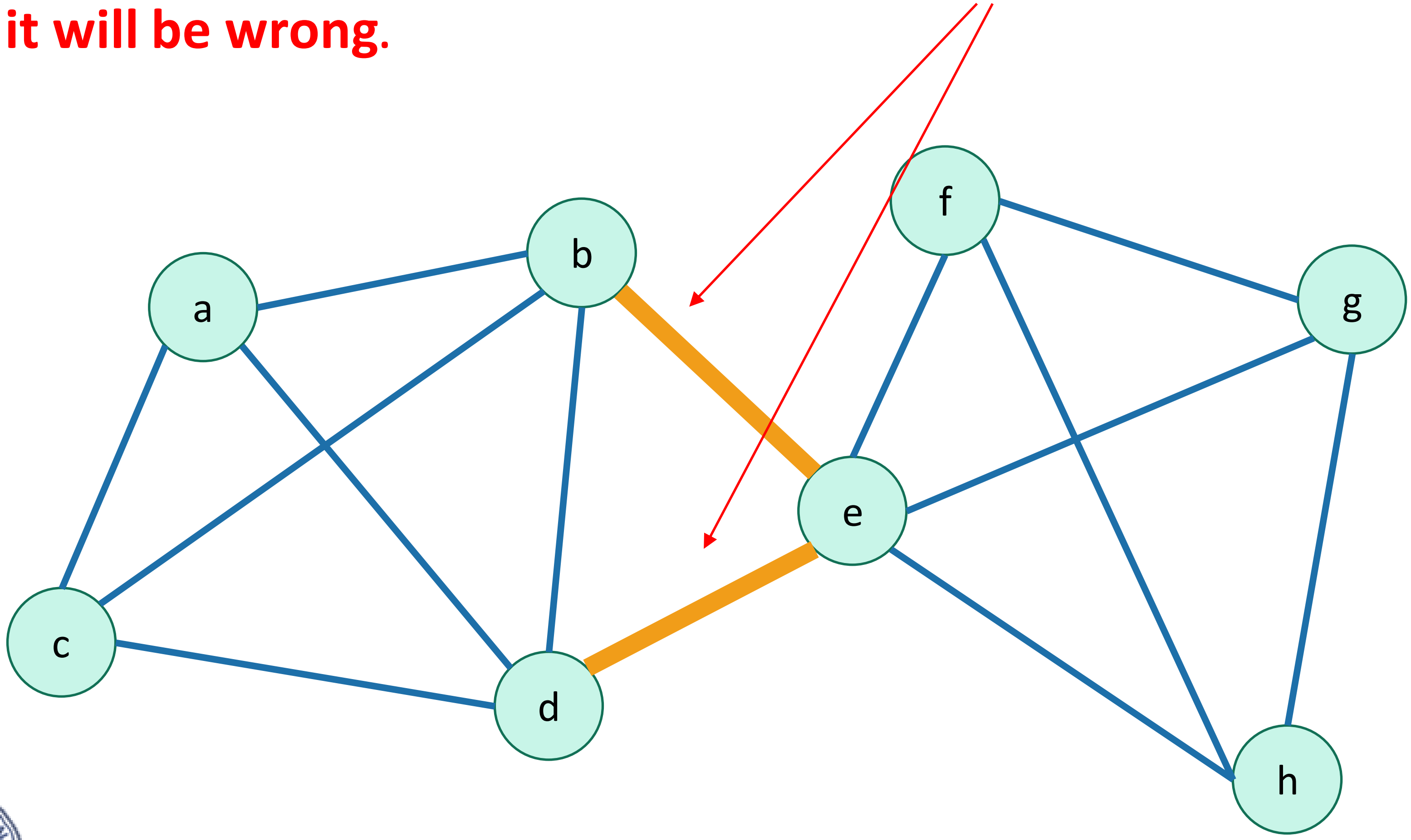
Say we had chosen this edge

Now there is **no way** we could return a cut that separates b and e.

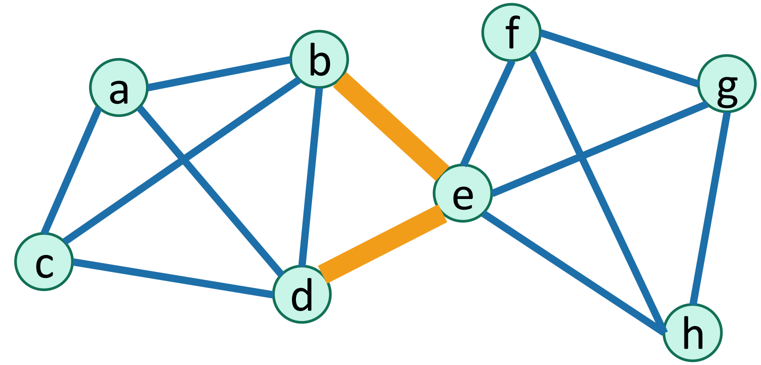


Even worse

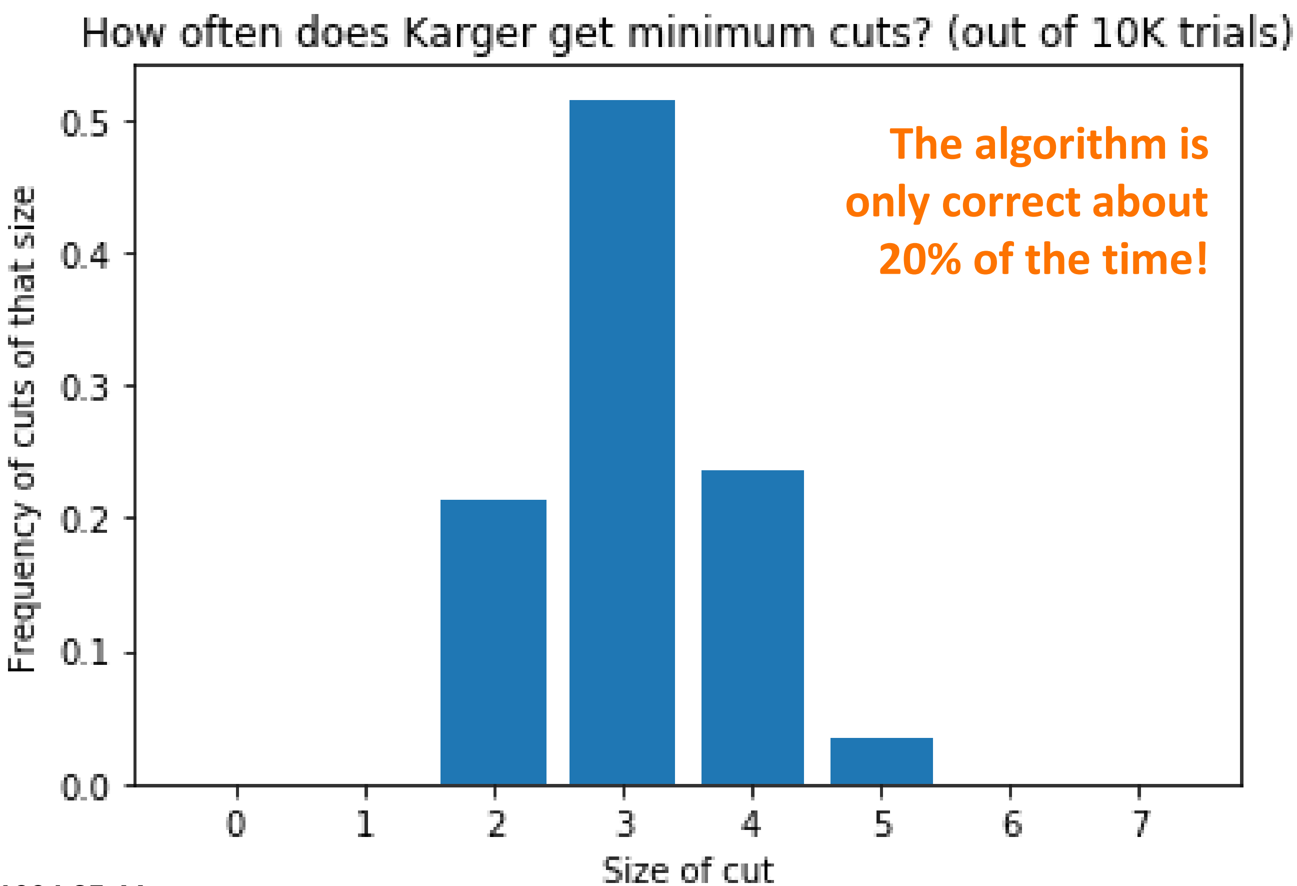
If the algorithm **EVER** chooses either of **these edges**,
it will be wrong.



How likely is that?

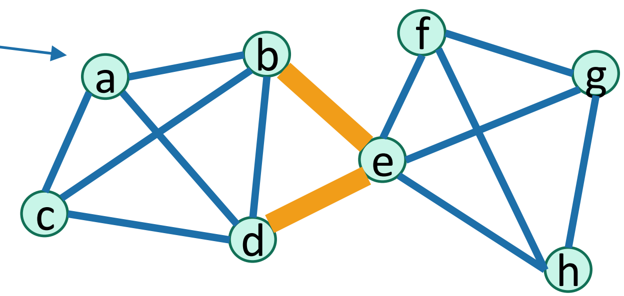


- For this particular graph, if do this 10,000 times:



That doesn't sound good

- To see why it's good after all, we'll do a case study of this graph.



The plan:

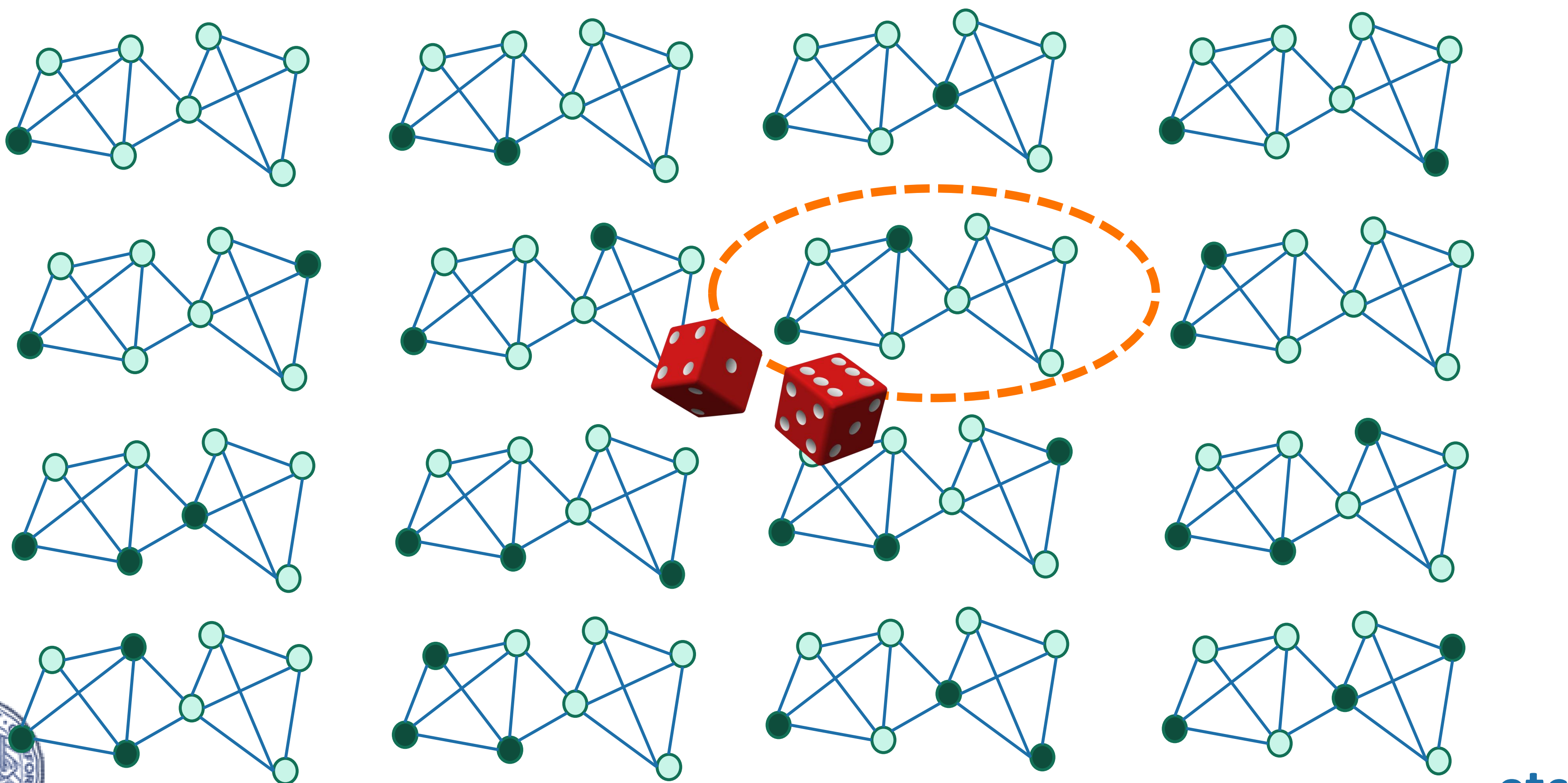
- See that 20% chance of correctness is actually nontrivial.
- Use repetition to boost an algorithm that's correct 20% of the time to an algorithm that's correct 99% of the time.

- To see the first point, let's compare Karger's algorithm to the algorithm:

Choose a completely random cut and hope that it's a minimum cut.

Uniformly random cut

- Pick a random way to split the vertices into two parts:



Uniformly random cut

- Pick a random way to split the vertices into two parts:

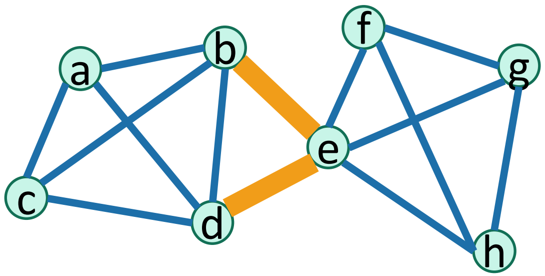
- The probability of choosing the minimum cut is*…

$$\frac{\text{number of min cuts in that graph}}{\text{number of ways to split 8 vertices in 2 parts}} = \frac{2}{2^8 - 2} \approx 0.008$$

- Aka, we get a minimum cut 0.8% of the time.

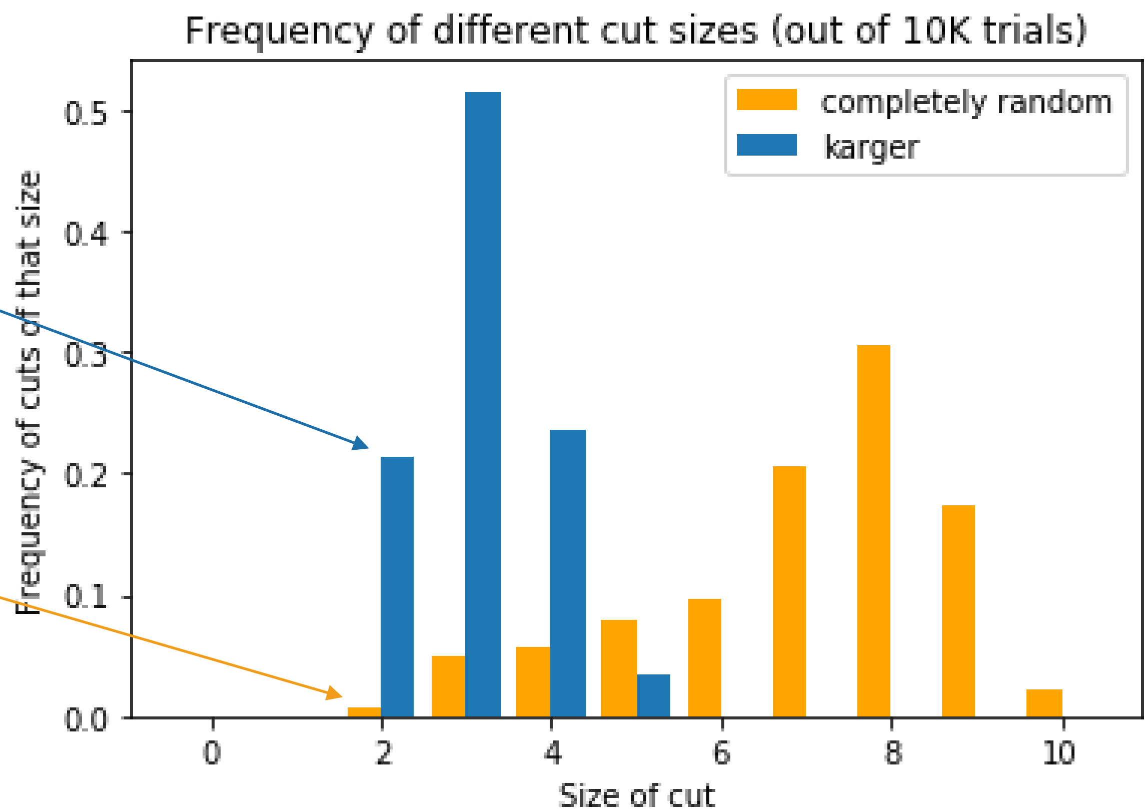
*For this example in particular

Karger is better than completely random!



Karger's alg. is correct about 20% of the time

Completely random is correct about 0.8% of the time

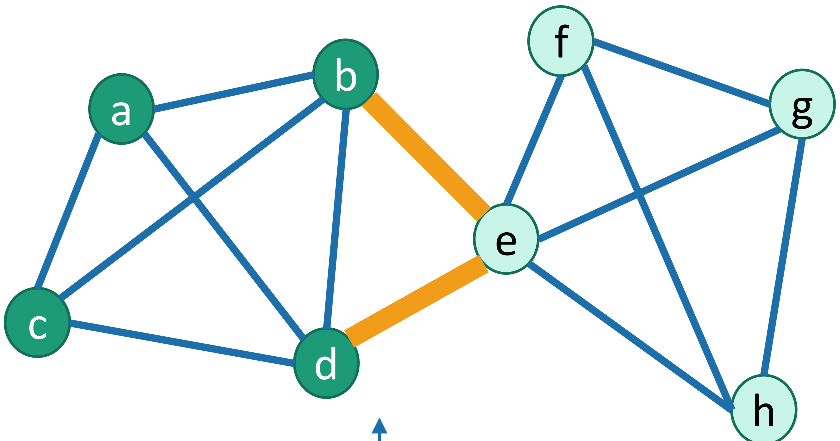


What’s going on?

Thing 1: It’s unlikely that Karger will hit the min cut since it’s so small!

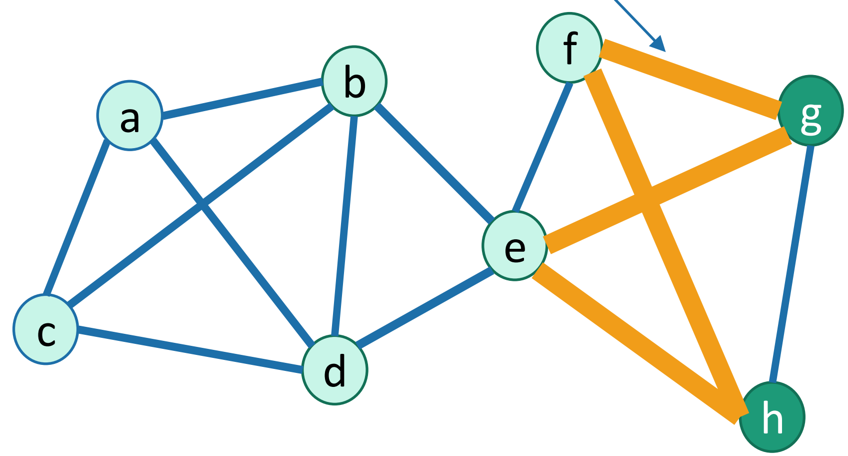
Lucky the lackadaisical lemur

- Which is more likely?



A: The algorithm never chooses either of the edges in **the minimum cut**.

B: The algorithm never chooses any of the edges in **this big cut**.

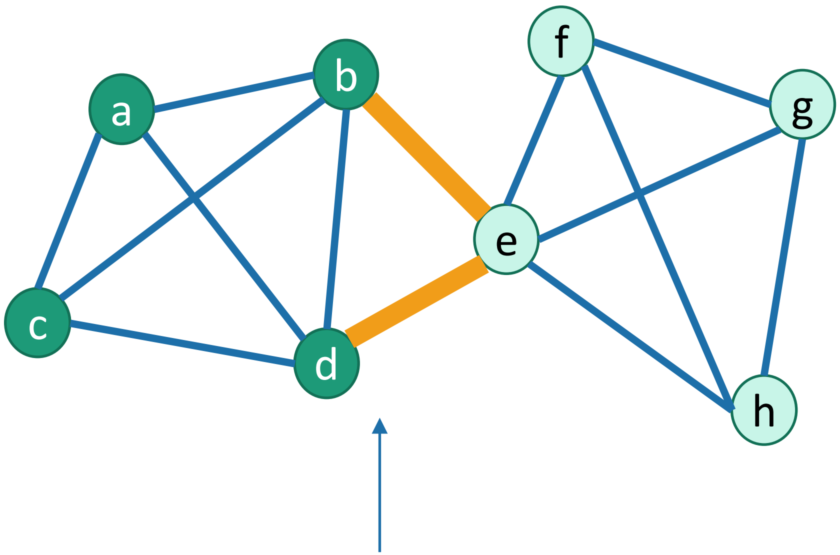


- Neither A nor B are very likely, **but A is more likely than B.**

What’s going on?

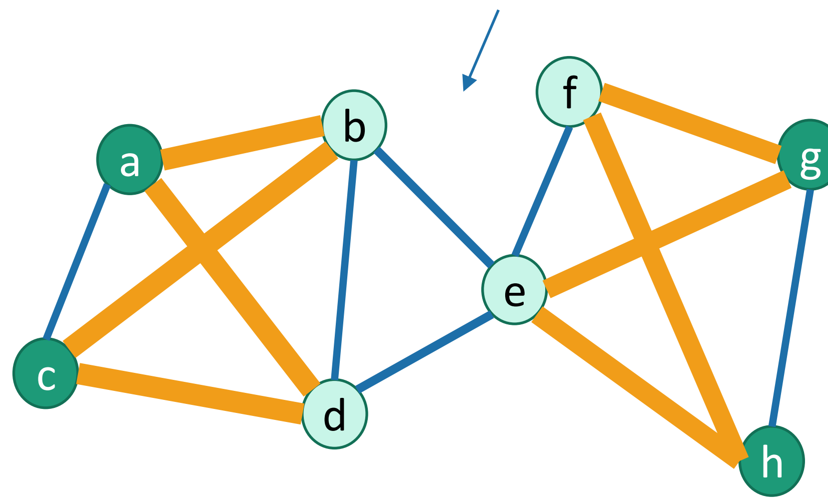
Thing 2: By only contracting edges we are ignoring certain really-not-minimal cuts.

Lucky the
lackadaisical lemur



A: This cut can be returned by Karger’s algorithm.

B: This cut can’t be returned by Karger’s algorithm!
(Because how would a and g end up in the same super-node?)



This cut actually separates the graph into three pieces, so it’s not minimal – either half of it is a smaller cut.

Why does that help?

- Okay, so it's better than completely random…
- We're still wrong about 80% of the time.
- The main idea: **repeat!**
  - If I'm wrong 20% of the time, then if I repeat it a few times I'll eventually get it right.

The plan:

- See that 20% chance of correctness is actually nontrivial.
- Use repetition to boost an algorithm that's correct 20% of the time to an algorithm that's correct 99% of the time.

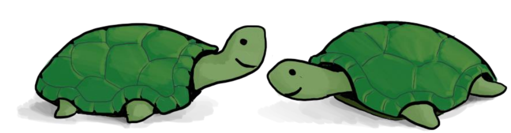
Thought experiment

- Suppose you have a magic button that produces one of 5 numbers, {a,b,c,d,e}, uniformly at random when you push it.
- You don’t know what {a,b,c,d,e} are.
- Q: What is the minimum of a,b,c,d,e?

6 3 3 2 5 2 5

How many times do you have to push the button, in expectation, before you see the minimum value?

What is the probability that you have to push it more than 5 times? 10 times?



Let's calculate the probabilities

This is the same calculation we've done a bunch of times:

- E[Number of times we push the button until we get the minimum value] = 1/(0.20) = 5

This one we've done less frequently:

- Pr[We push the button t times and don't ever get the min] = $(1 - 0.2)^t$
- Pr[We push the button 5 times and don't ever get the min] = $(1 - 0.2)^5 \approx 0.33$
- Pr[We push the button 10 times and don't ever get the min] = $(1 - 0.2)^{10} \approx 0.1$