BLG 336E

Analysis of Algorithms II

Lecture 10:

Network Flow I

Min Cut and Karger's Algorithm

Last time

Dynamic Programming!

- Dynamic programming is an algorithm design paradigm.
- Basic idea:
  - Identify **optimal sub-structure**
    - Optimum to the big problem is built out of optima of small sub-problems
  - Take advantage of **overlapping sub-problems**
    - Only solve each sub-problem once, then use it again and again
  - Keep track of the solutions to sub-problems in a table as you build to the final solution.

Recap

- We saw examples of how to come up with dynamic programming algorithms.
  - Longest Common Subsequence
  - Knapsack two ways
  - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Knapsack problem

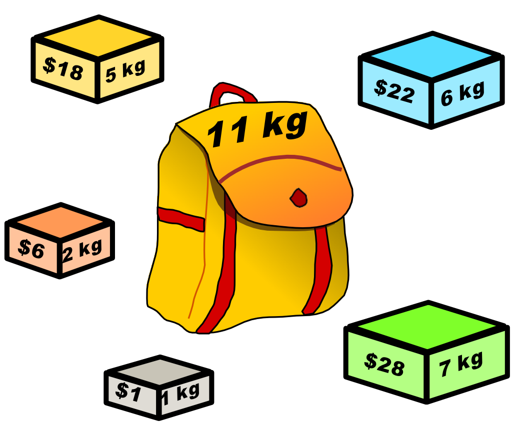
Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.

Ex. The subset $\{ 1, 2, 5 \}$ has value \$35 (and weight 10).

Ex. The subset $\{ 3, 4 \}$ has value \$40 (and weight 11).

Assumption. All values and weights are integral.



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

weights and values
can be arbitrary
positive integers

knapsack instance
(weight limit $W = 11$)

Dynamic programming: quiz 2

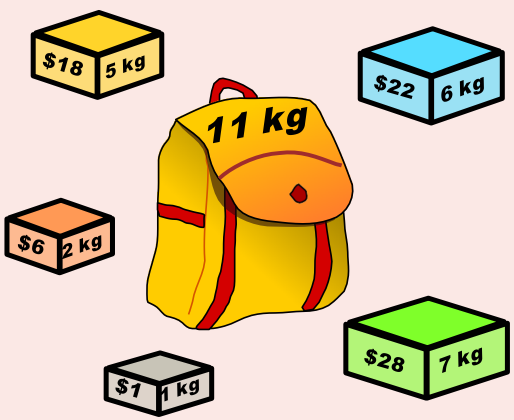
Which algorithm solves knapsack problem?

A. Greedy-by-value: repeatedly add item with maximum v_i.

B. Greedy-by-weight: repeatedly add item with minimum w_i.

C. Greedy-by-ratio: repeatedly add item with maximum ratio v_i / w_i.

D. None of the above. 



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit $W = 11$)

Dynamic programming: quiz 3

Which subproblems?

A. $OPT(w)$ = optimal value of knapsack problem with weight limit w.

B. $OPT(i)$ = optimal value of knapsack problem with items $1, \ldots, i$.

C. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$ subject to weight limit w. 

D. Any of the above.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

possibly because $w_i > w$

Case 1. $OPT(i, w)$ does not select item i.

- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ subject to weight limit w.

optimal substructure property
(proof via exchange argument)

Case 2. $OPT(i, w)$ selects item i.

- Collect value v_i.
- New weight limit $= w - w_i$.
- $OPT(i, w)$ selects best of $\{ 1, 2, \ldots, i-1 \}$ subject to new weight limit.

Bellman equation.

$$OPT(i,w) \;=\; \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\;\; v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming

KNAPSACK(n, W, w_1, …, w_n, v_1, …, v_n)

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

    IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

    ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w - w_i] \}.$

previously computed values

RETURN $M[n, W]$.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up dynamic programming demo

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

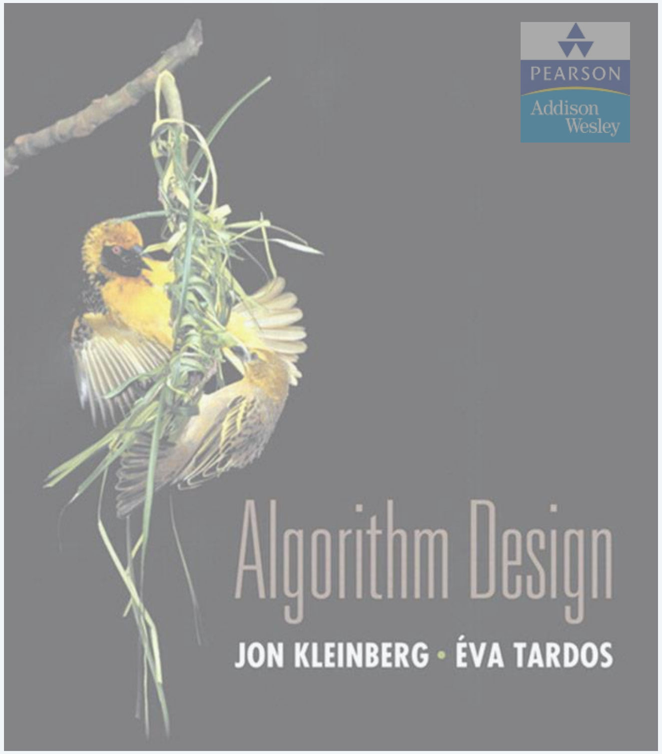
$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = optimal value of knapsack problem with items 1, …, i, subject to weight limit w

6. Dynamic Programming I



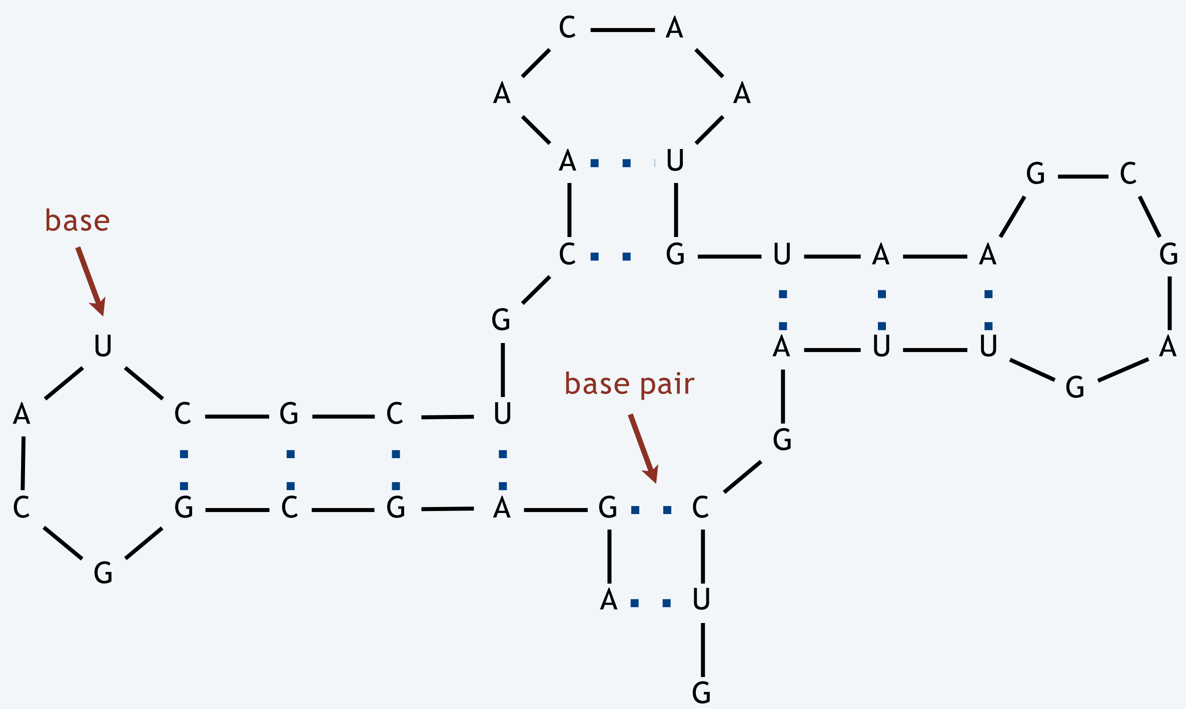
Section 6.5

- *weighted interval scheduling*
- *segmented least squares*
- *knapsack problem*
- *RNA secondary structure*

RNA secondary structure

RNA. String $B = b_1 b_2 \ldots b_n$ over alphabet $\{ A, C, G, U \}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

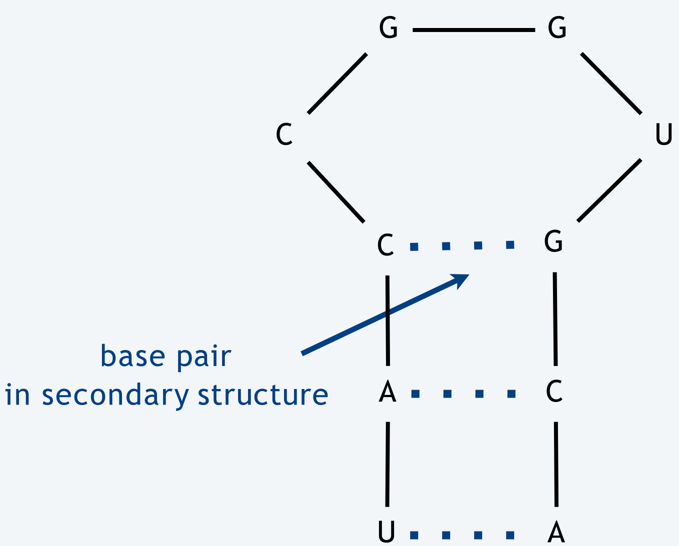


RNA secondary structure for GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA

RNA secondary structure

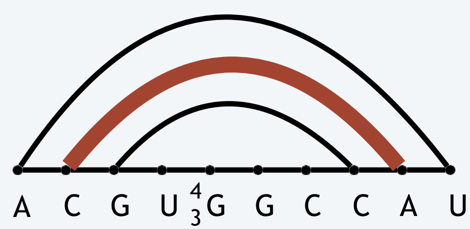
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.



B = ACGUGGCCCAU

$S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$

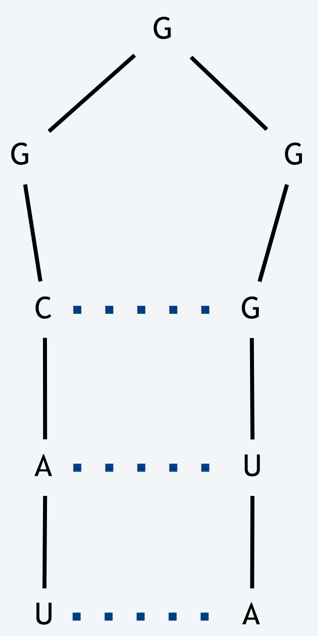


S is not a secondary structure
(C-A is not a valid Watson-Crick pair)

RNA secondary structure

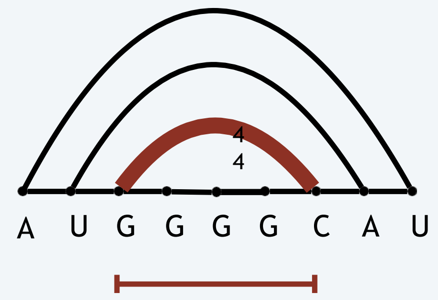
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.



B = AUGGGCAU

$S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$



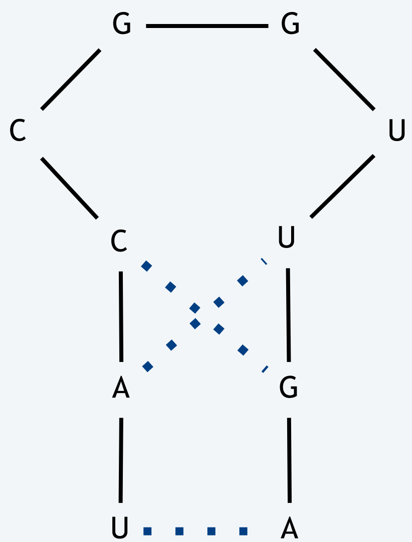
S is not a secondary structure

(≤4 intervening bases between G and C)

RNA secondary structure

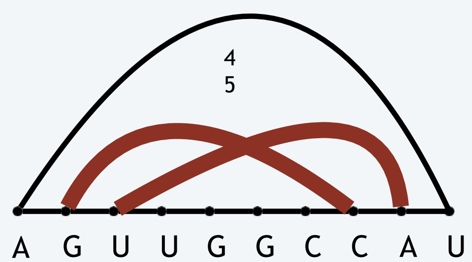
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.



B = ACUUGGCCAU

$S = \{ (b_1, b_{10}), (b_2, b_8), (b_3, b_9) \}$

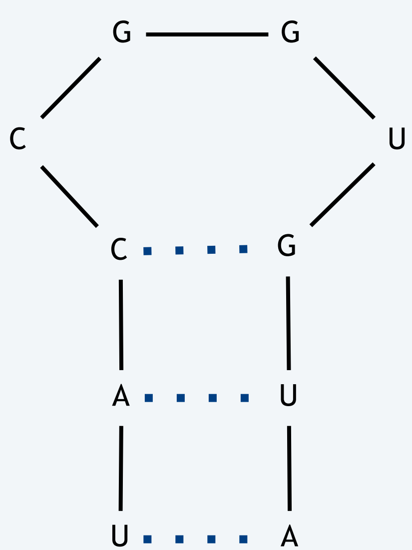


S is not a secondary structure (G-C and U-A cross)

RNA secondary structure

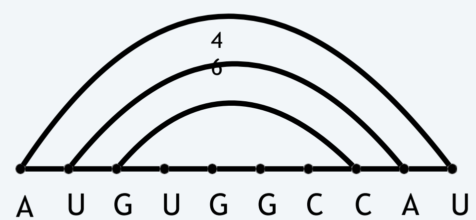
Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.



$B = \text{AUGUGGCCAU}$

$S = \{ (b_1, b_{10}), (b_2, b_9), (b_3, b_8) \}$



S is a secondary structure (with 3 base pairs)

RNA secondary structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: A–U, U–A, C–G, or G–C.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S, then we cannot have $i < k < j < \ell$.

Free-energy hypothesis. RNA molecule will form the secondary structure with the minimum total free energy.

approximate by number of base pairs
(more base pairs $\Rightarrow$ lower free energy)

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

Dynamic programming: quiz 5

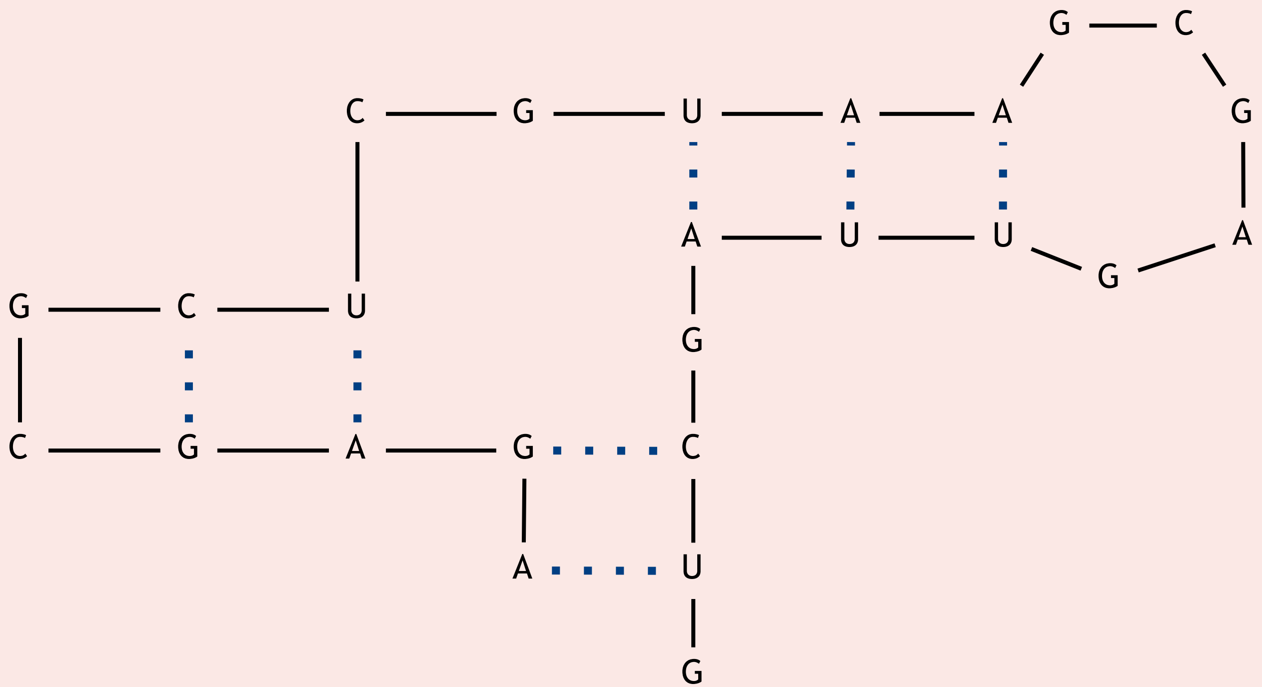
Is the following a secondary structure?

A. Yes.

B. No, violates Watson–Crick condition.

C. No, violates no-sharp-turns condition. 

D. No, violates no-crossing condition.



Dynamic programming: quiz 6

Which subproblems?

A. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_1 b_2 \ldots b_j$.

B. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_j b_{j+1} \ldots b_n$.

C. Either A or B.

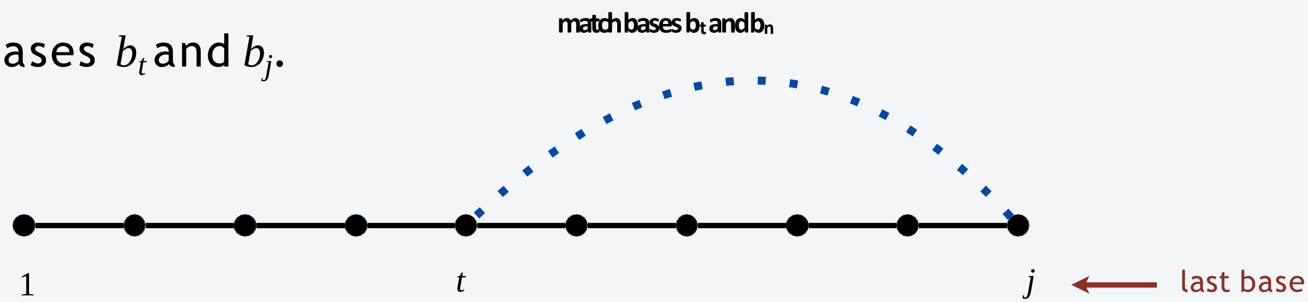
D. Neither A nor B.

RNA secondary structure: subproblems

First attempt. $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_j$.

Goal. $OPT(n)$.

Choice. Match bases b_t and b_j.



Difficulty. Results in two subproblems (but one of wrong form).

- Find secondary structure in $b_1 b_2 \ldots b_{t-1}$. ← $OPT(t-1)$
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{j-1}$. ← need more subproblems (first base no longer b_1)

Dynamic programming over intervals

Def. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

Case 1. If $i \geq j - 4$.

- $OPT(i, j) = 0$ by no-sharp-turns condition.

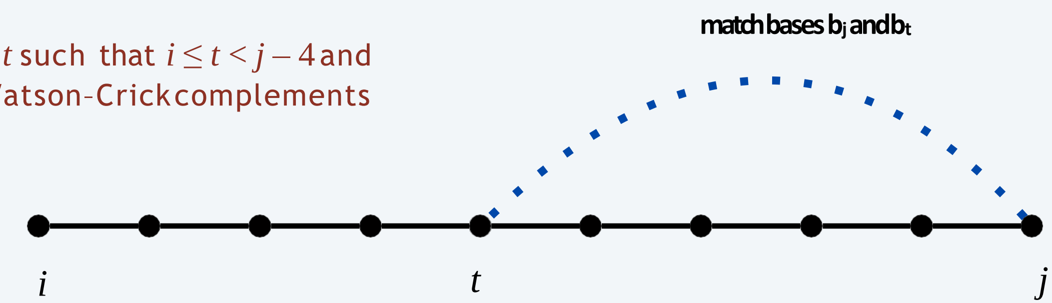
Case 2. Base b_j is not involved in a pair.

- $OPT(i, j) = OPT(i, j - 1)$.

Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.

- Non-crossing condition decouples resulting two subproblems.
- $OPT(i, j) = 1 + \max_t \{ OPT(i, t - 1) + OPT(t + 1, j - 1) \}$.

take max over t such that $i \leq t < j - 4$ and b_t and b_j are Watson-Crick complements



In which order to compute $OPT(i, j)$?

A. Increasing i, then j.

B. Increasing j, then i. 

C. Either A or B.

D. Neither A nor B.

Bottom-up dynamic programming over intervals

Q. In which order to solve the subproblems?

A. Do shortest intervals first—increasing order of $|j - i|$.

RNA-SECONDARY-STRUCTURE$(n, b_1, \ldots, b_n)$

FOR $k = 5$ TO $n - 1$

  FOR $i = 1$ TO $n - k$

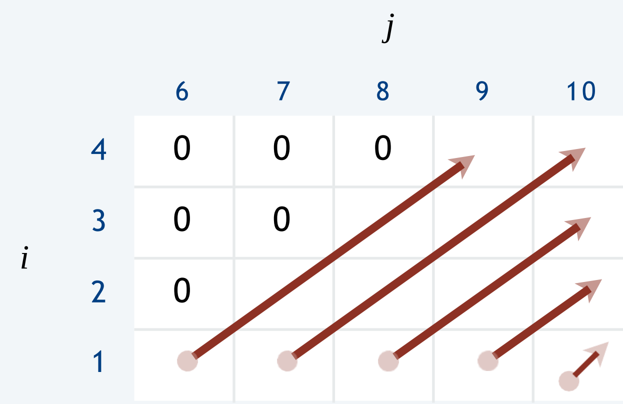
    $j \leftarrow i + k$.

    Compute $M[i, j]$ using formula.

RETURN $M[1, n]$.

---

all needed values are already computed



order in which to solve subproblems

5
3

Theorem. The DP algorithm solves the RNA secondary structure problem in $O(n^3)$ time and $O(n^2)$ space.

Dynamic programming summary

Outline.

typically, only a polynomial number of subproblems

- Define a collection of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from "smallest" to "largest" that enables determining a solution to a subproblem from solutions to smaller subproblems.

Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Intervals: RNA secondary structure.

Top-down vs. bottom-up dynamic programming. Opinions differ.

Today

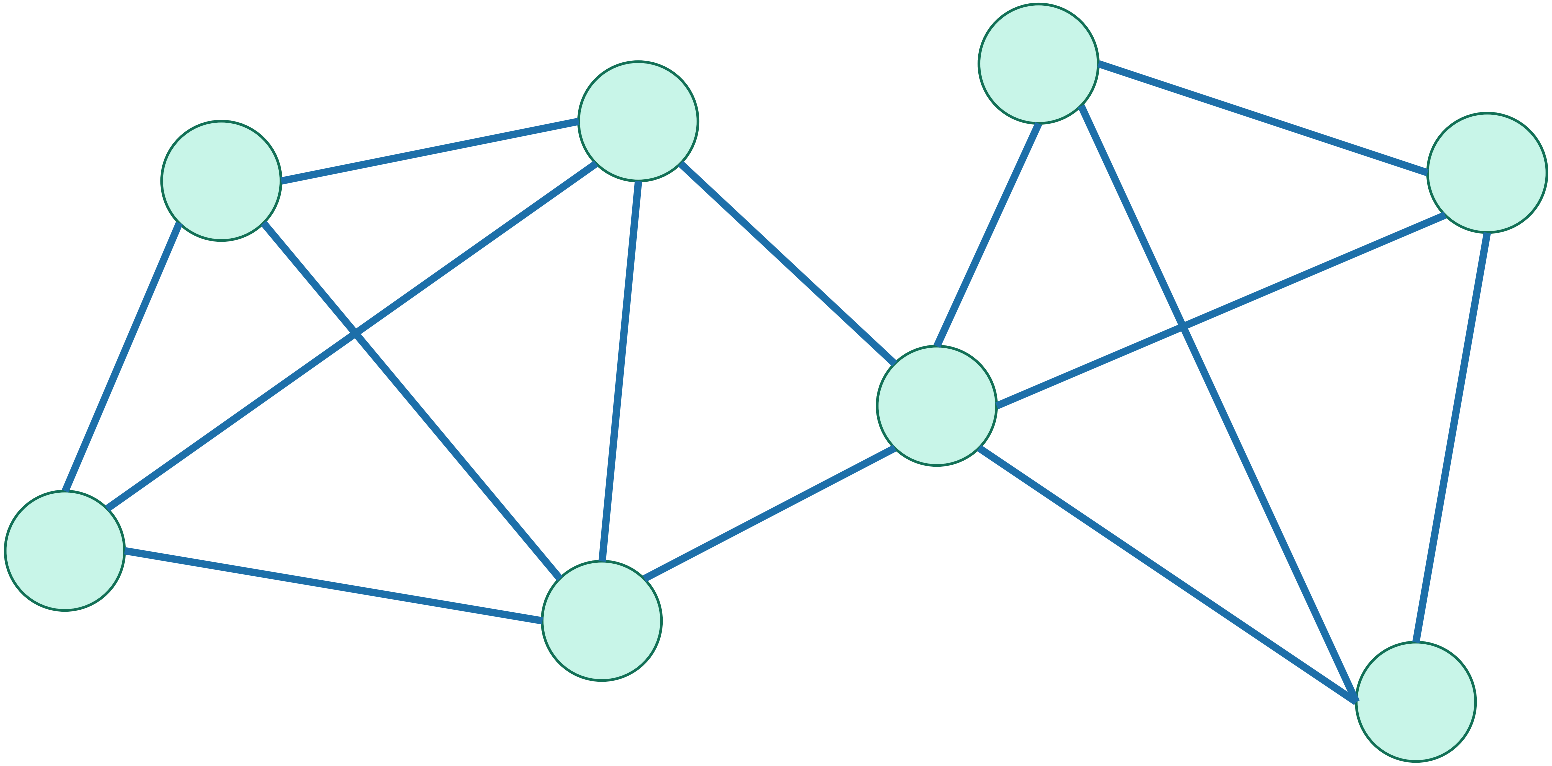
- Minimum Cuts!
  - Karger’s algorithm
  - Karger-Stein algorithm

  - Back to **randomized algorithms!**

Recall: cuts in graphs

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

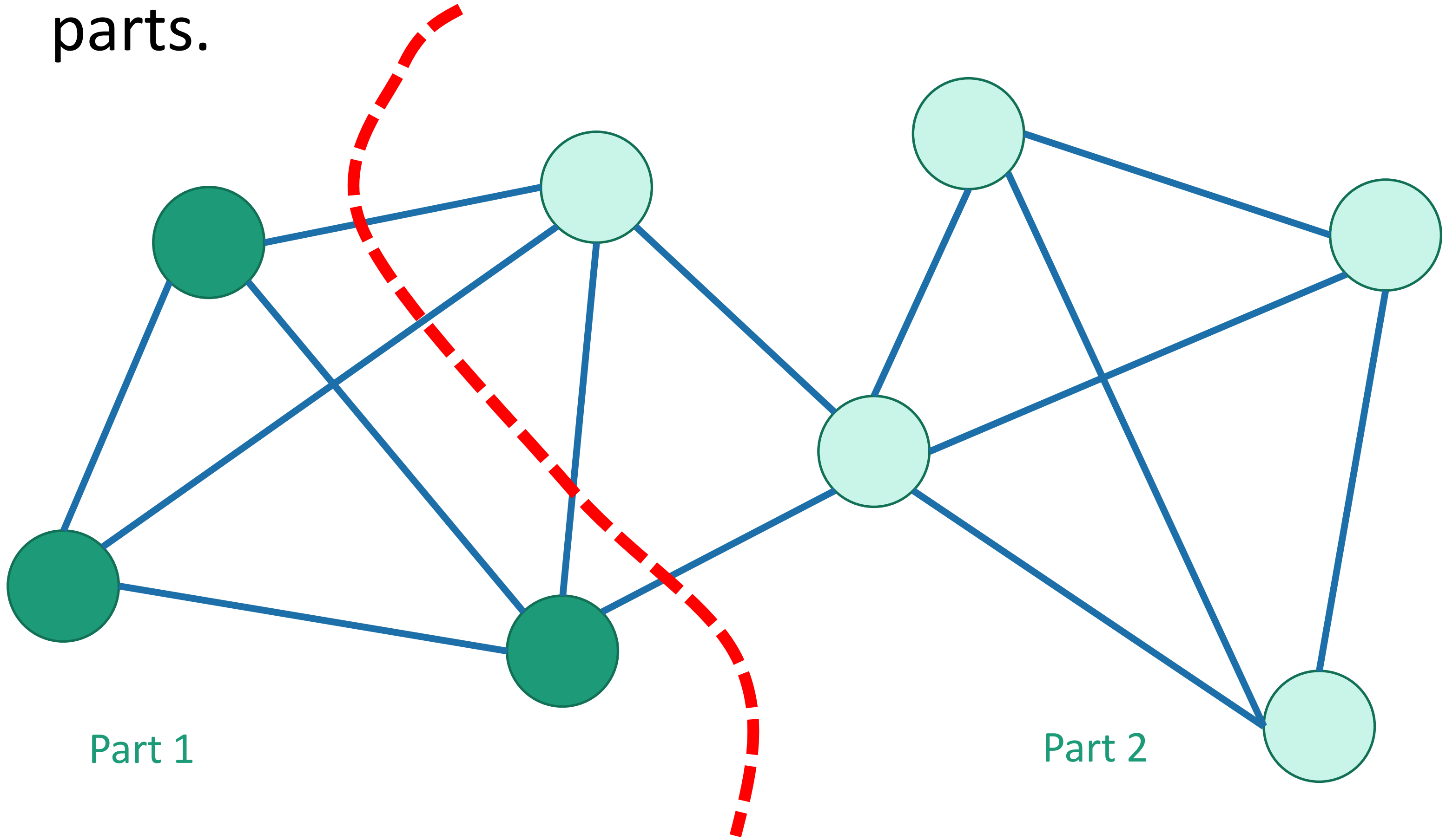
- A cut is a partition of the vertices into two nonempty parts.



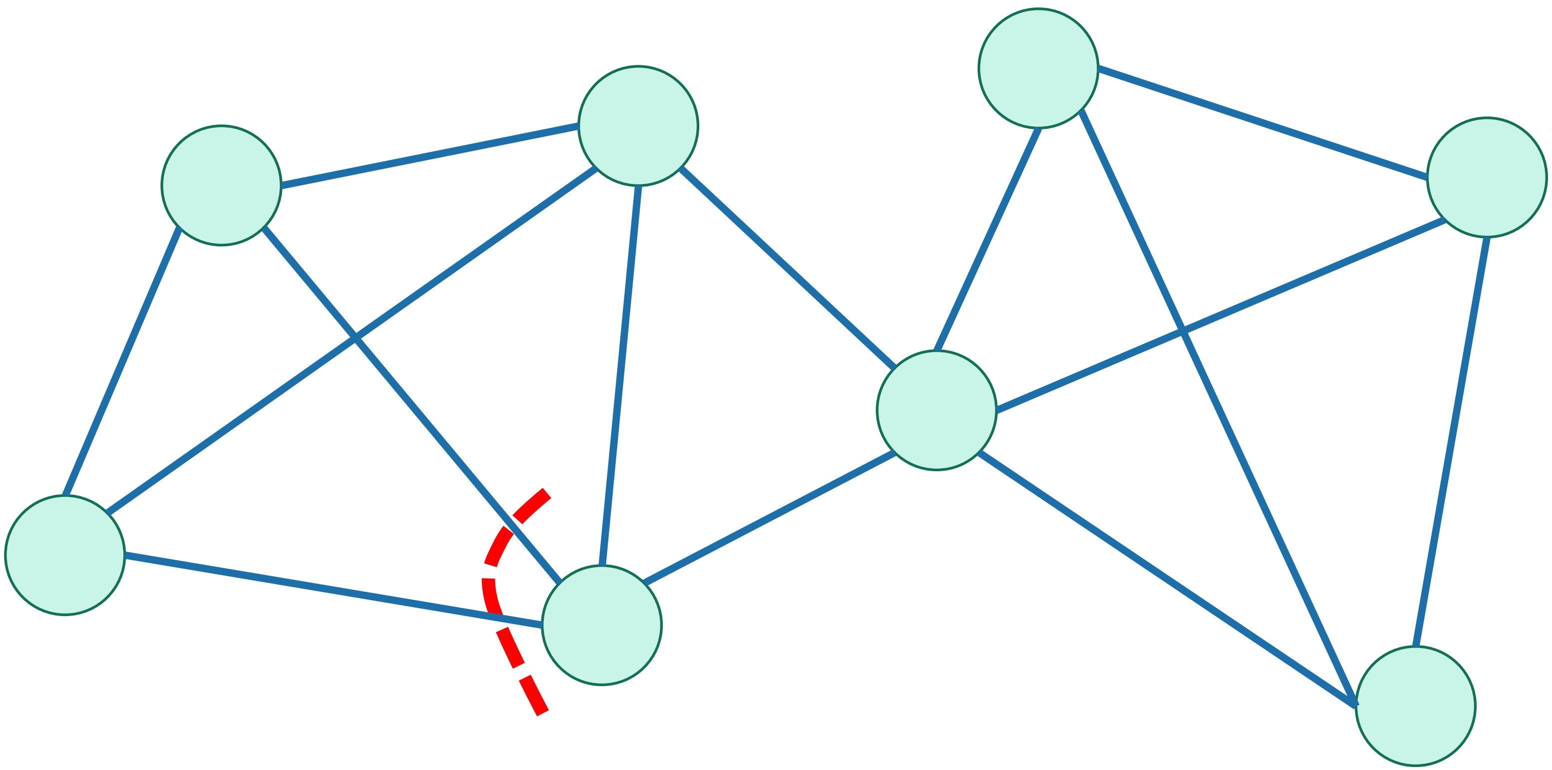
Recall: cuts in graphs

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

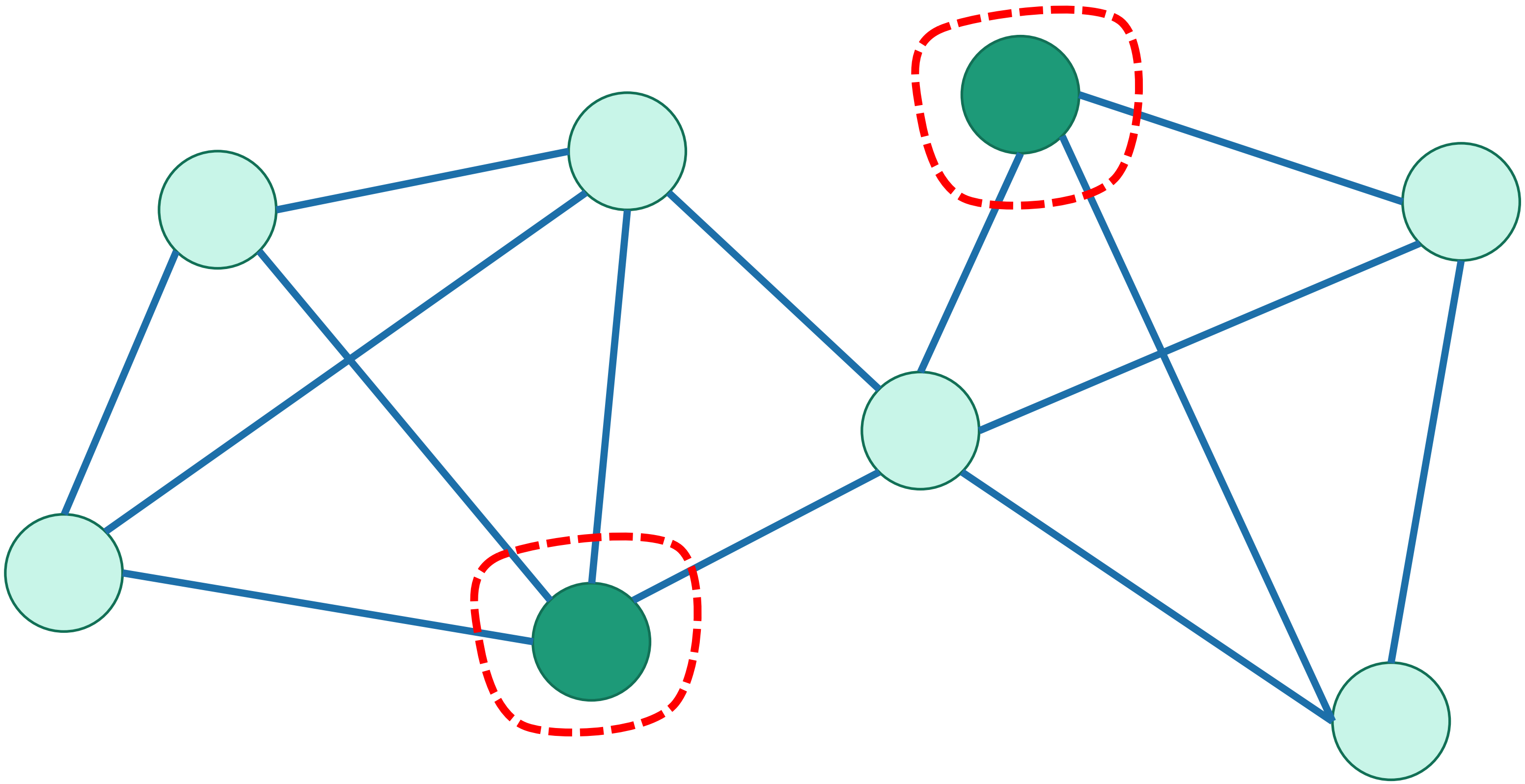
- A cut is a partition of the vertices into two nonempty parts.



This is not a cut



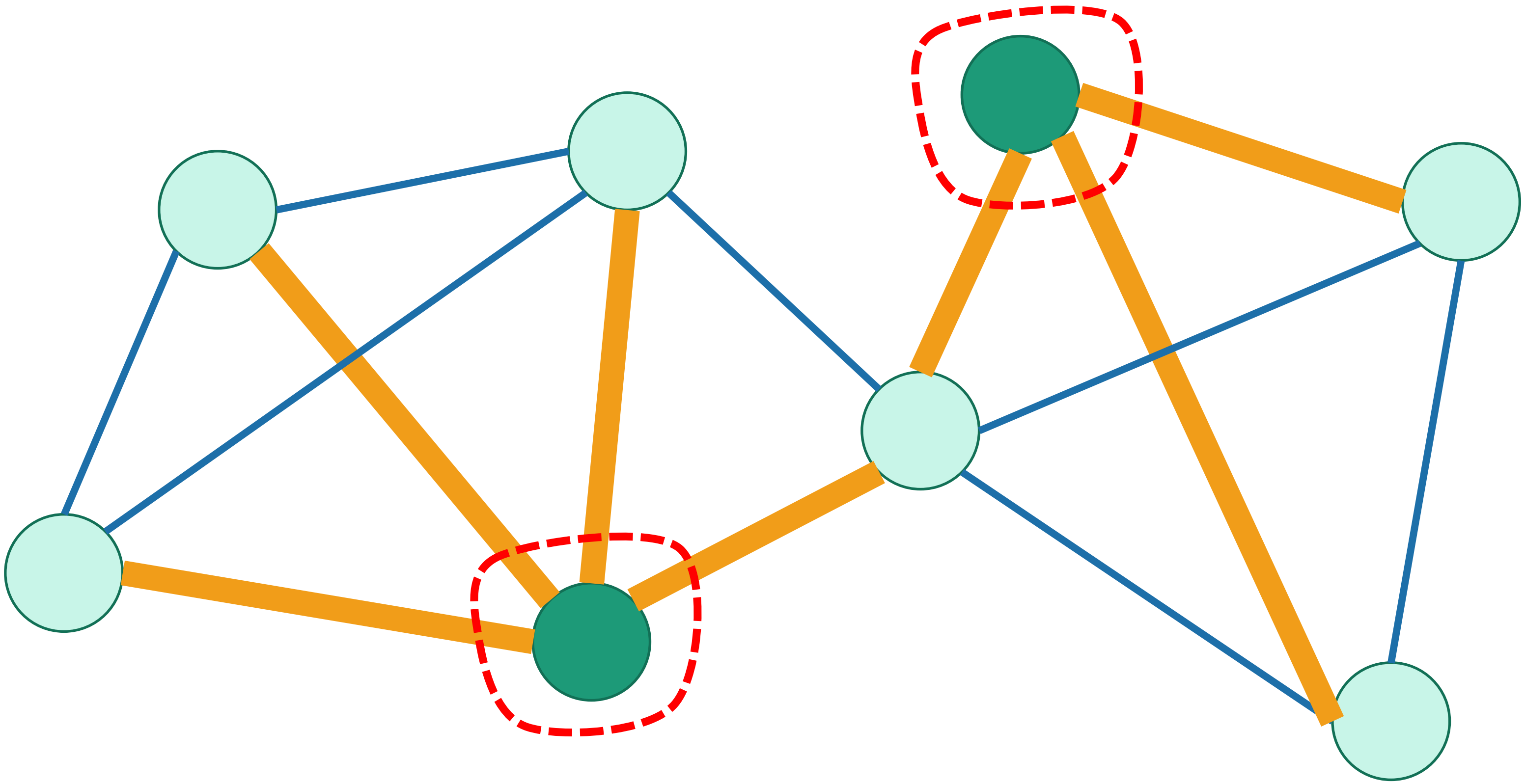
This is a cut



This is a cut

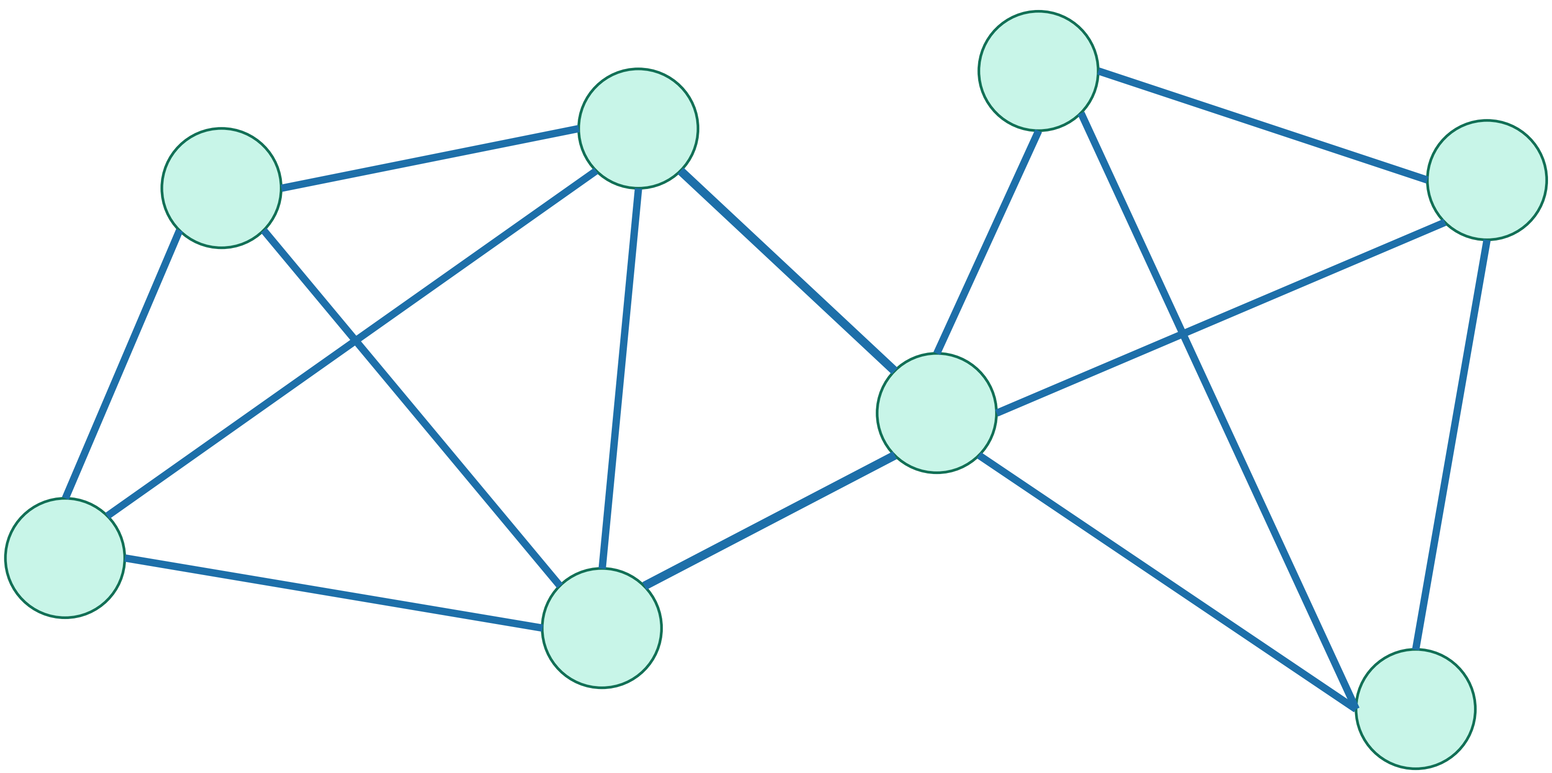
These edges **cross the cut.**

- They go from one part to the other.



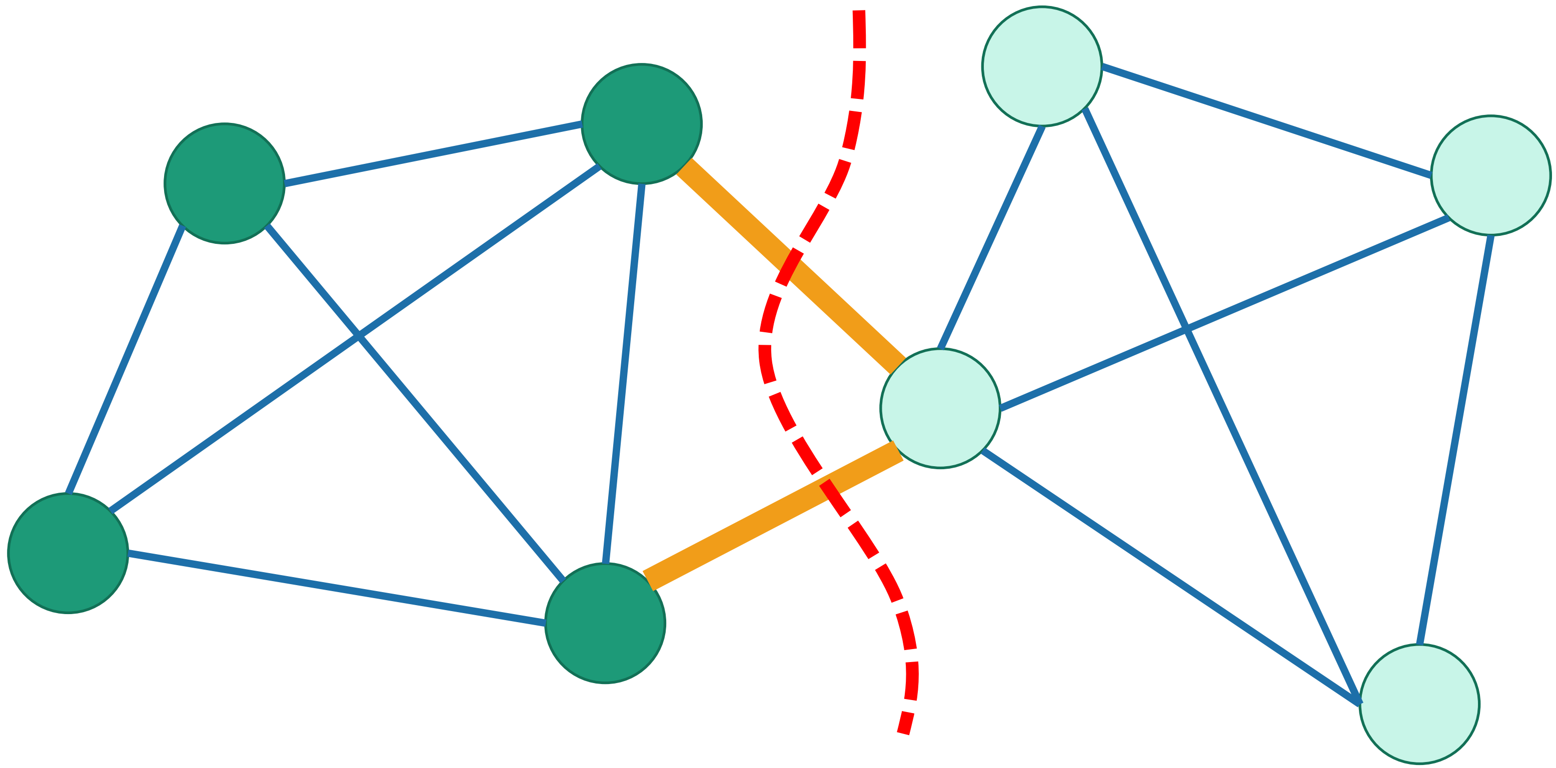
A (global) minimum cut

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



A (global) minimum cut

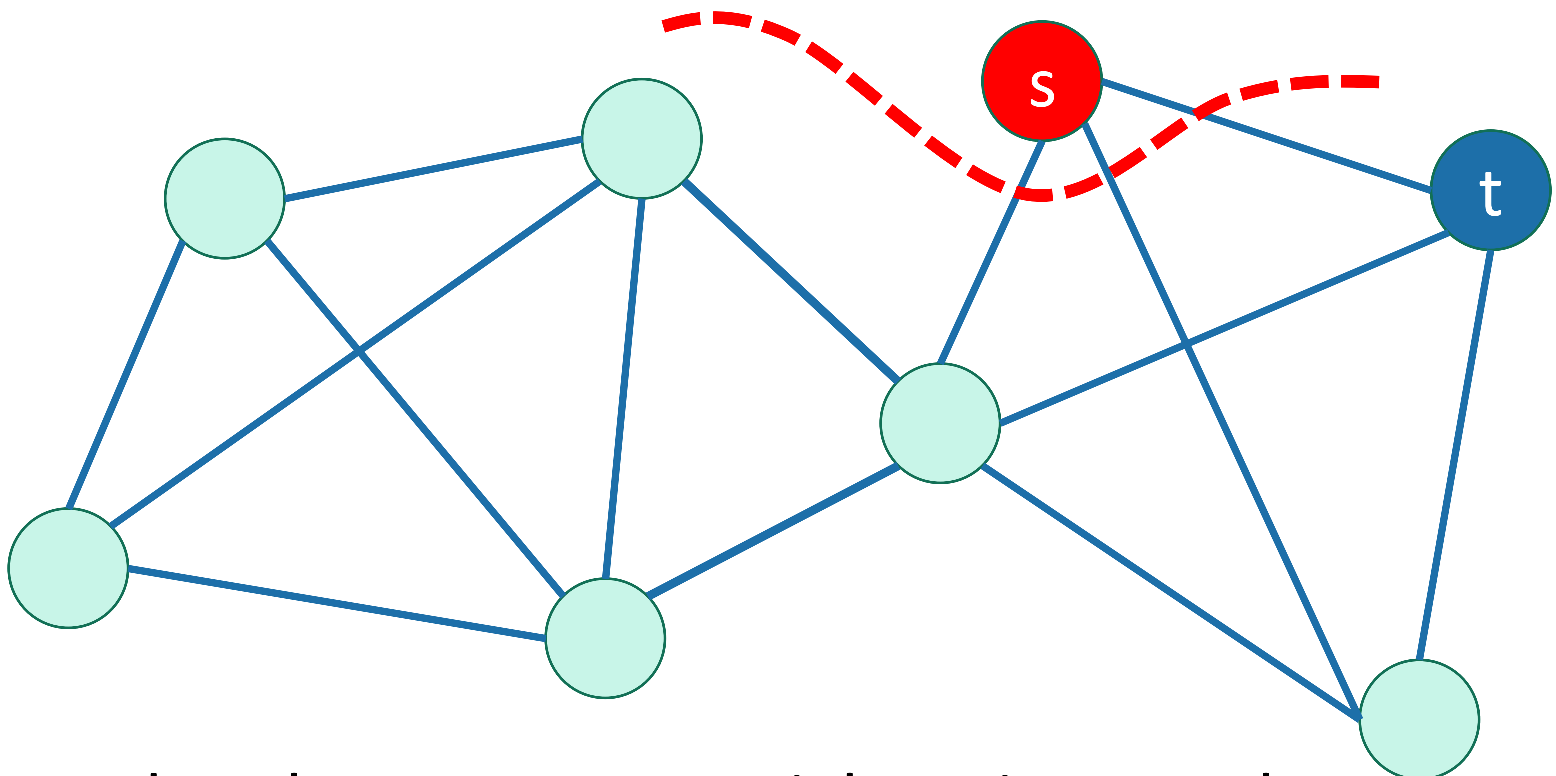
is a cut that has the fewest edges possible crossing it.



Why “global”?

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

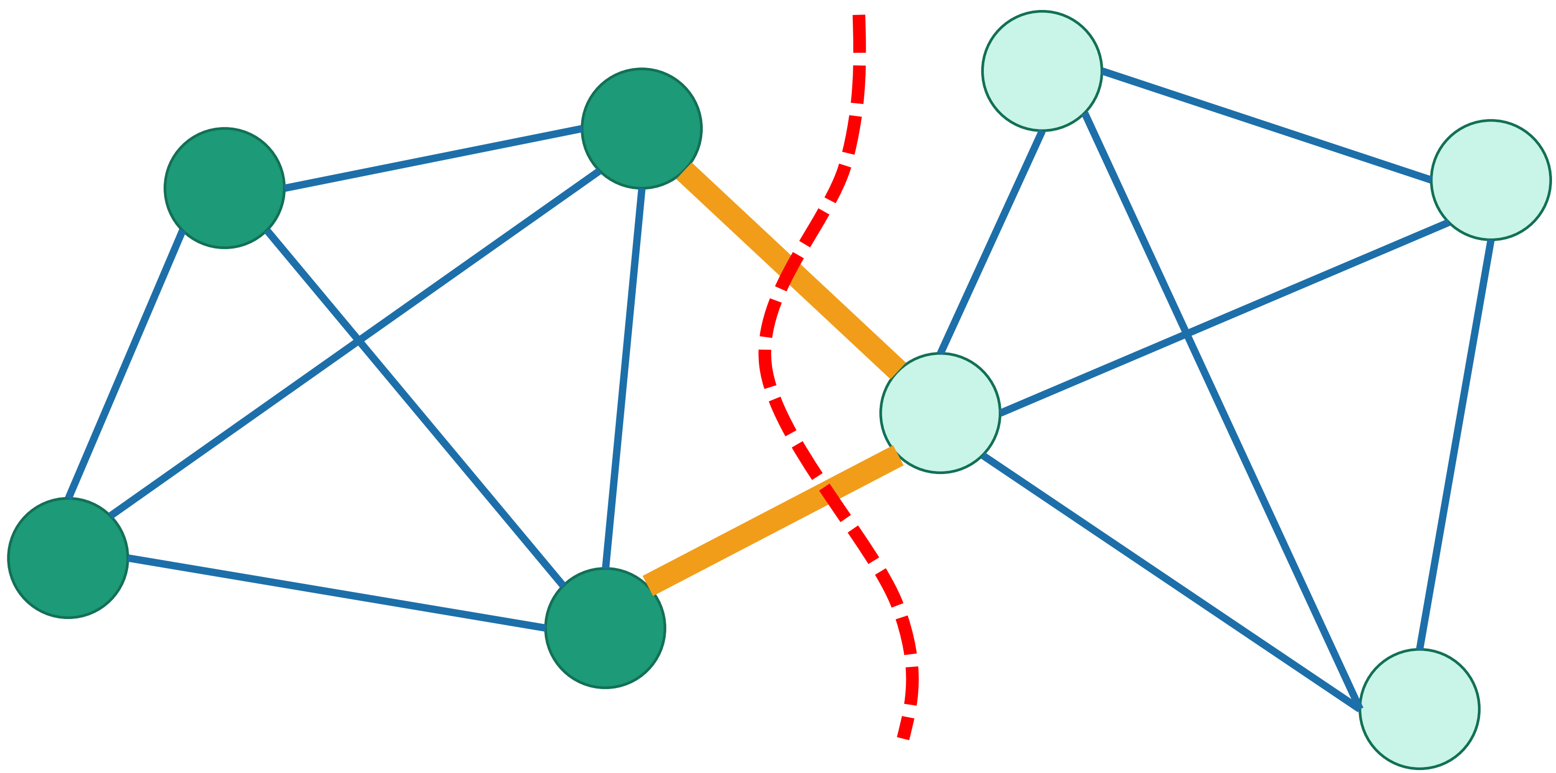
Minimum cut which separates a specified vertex s from t



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

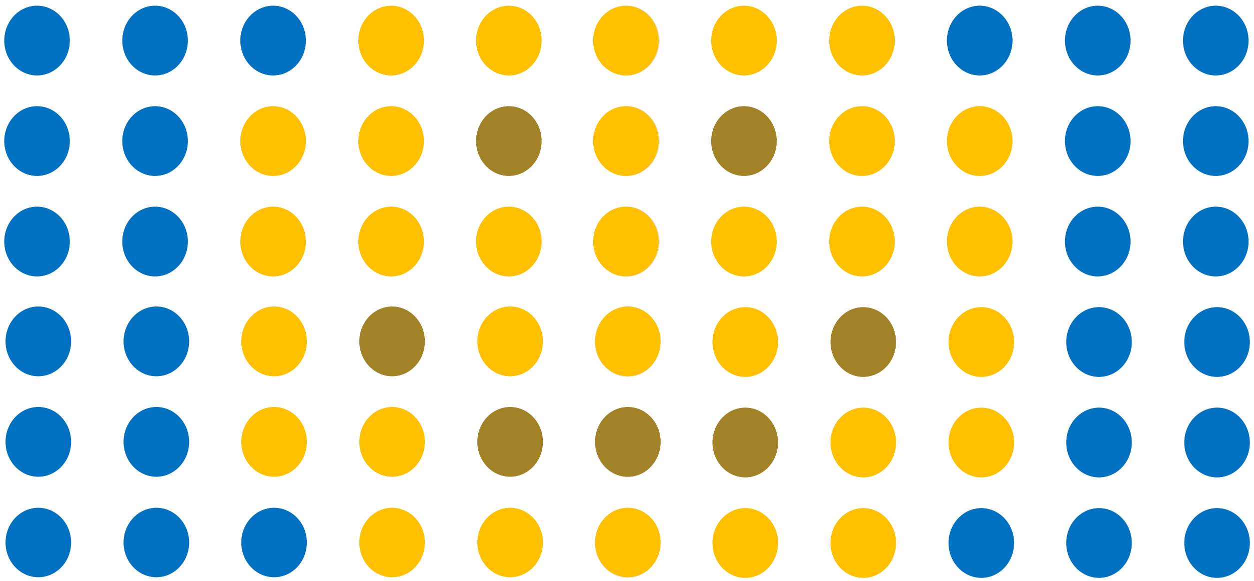
A (global) minimum cut

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



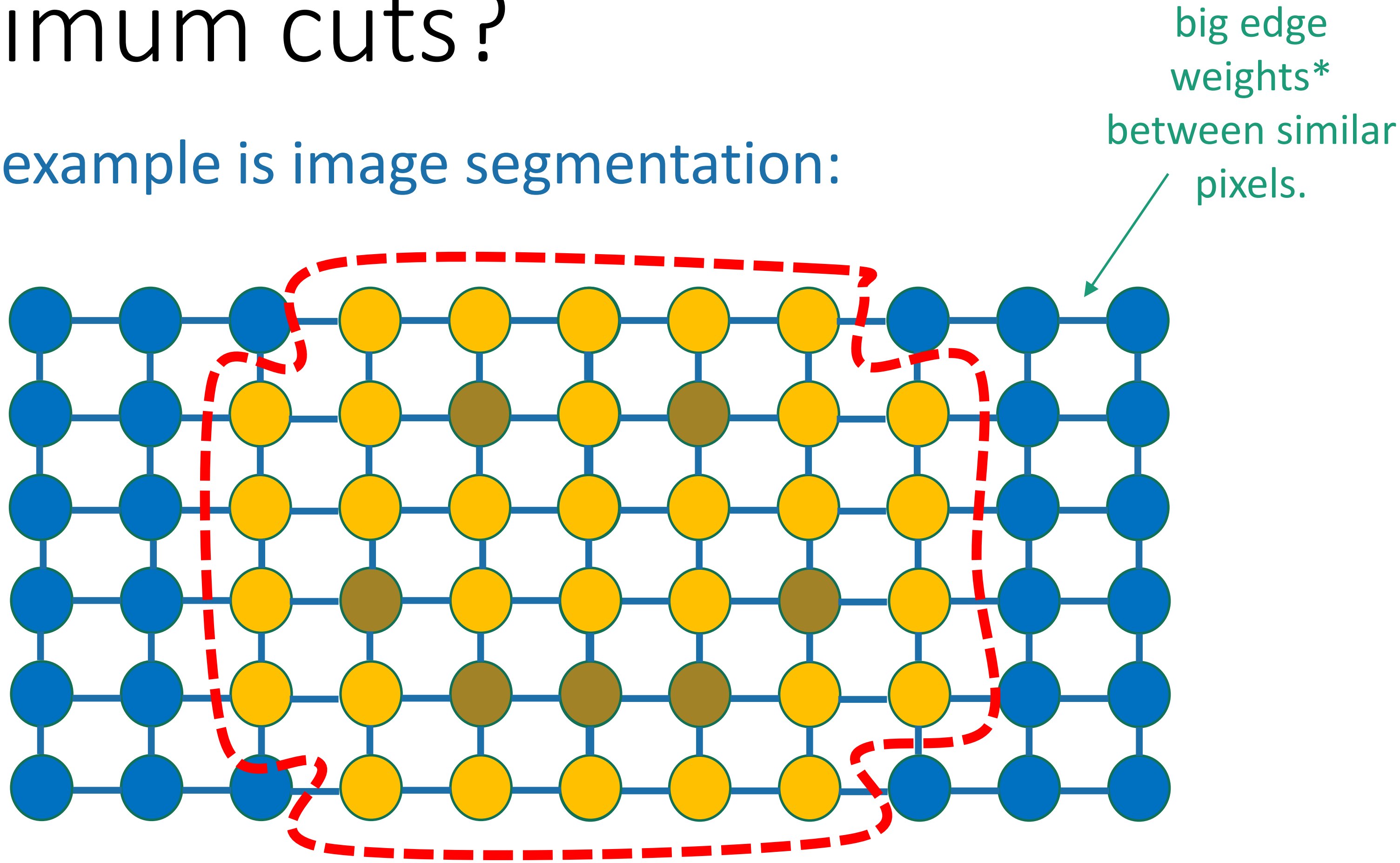
Why might we care about global minimum cuts?

- One example is image segmentation:



Why might we care about global minimum cuts?

- One example is image segmentation:



- We'll see more applications for other sorts of min-cuts next week

*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
- Karger’s algorithm **might be wrong**.
    - Compare to 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.

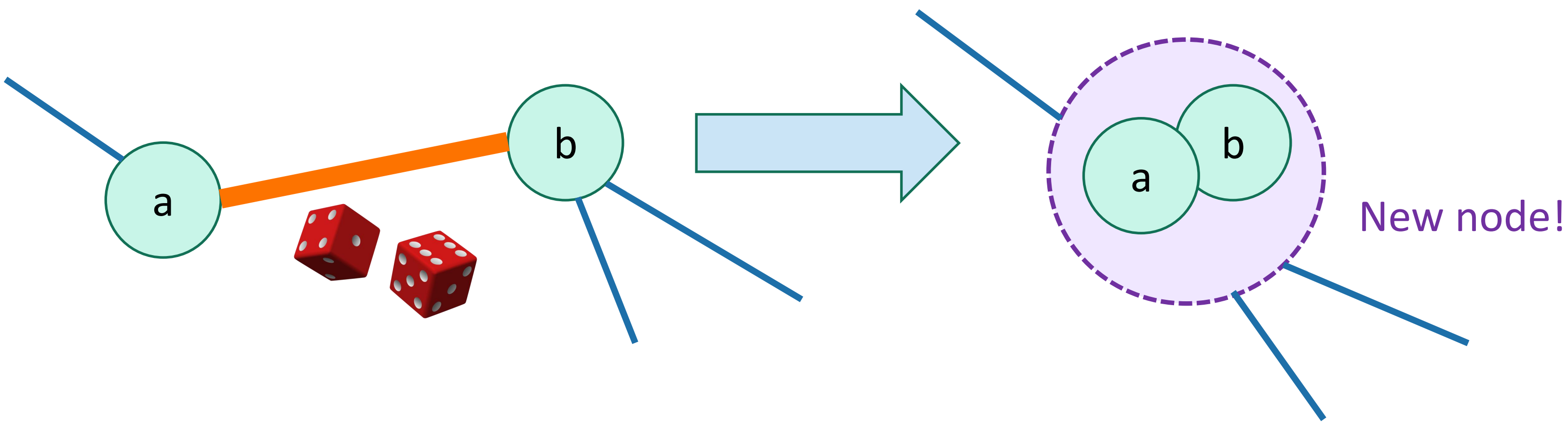
Formally:

- 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 ____.

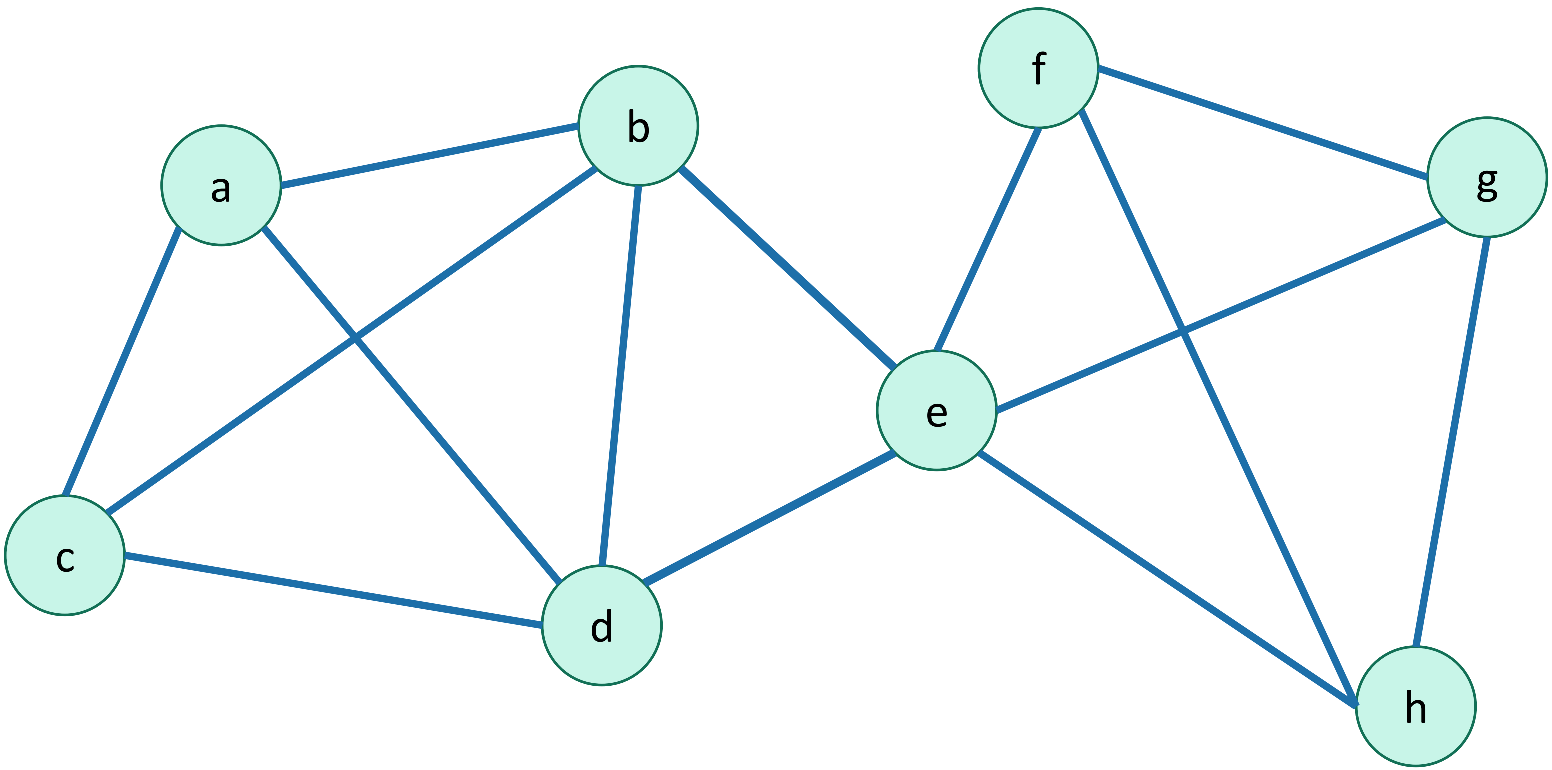
Karger's Algorithm

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

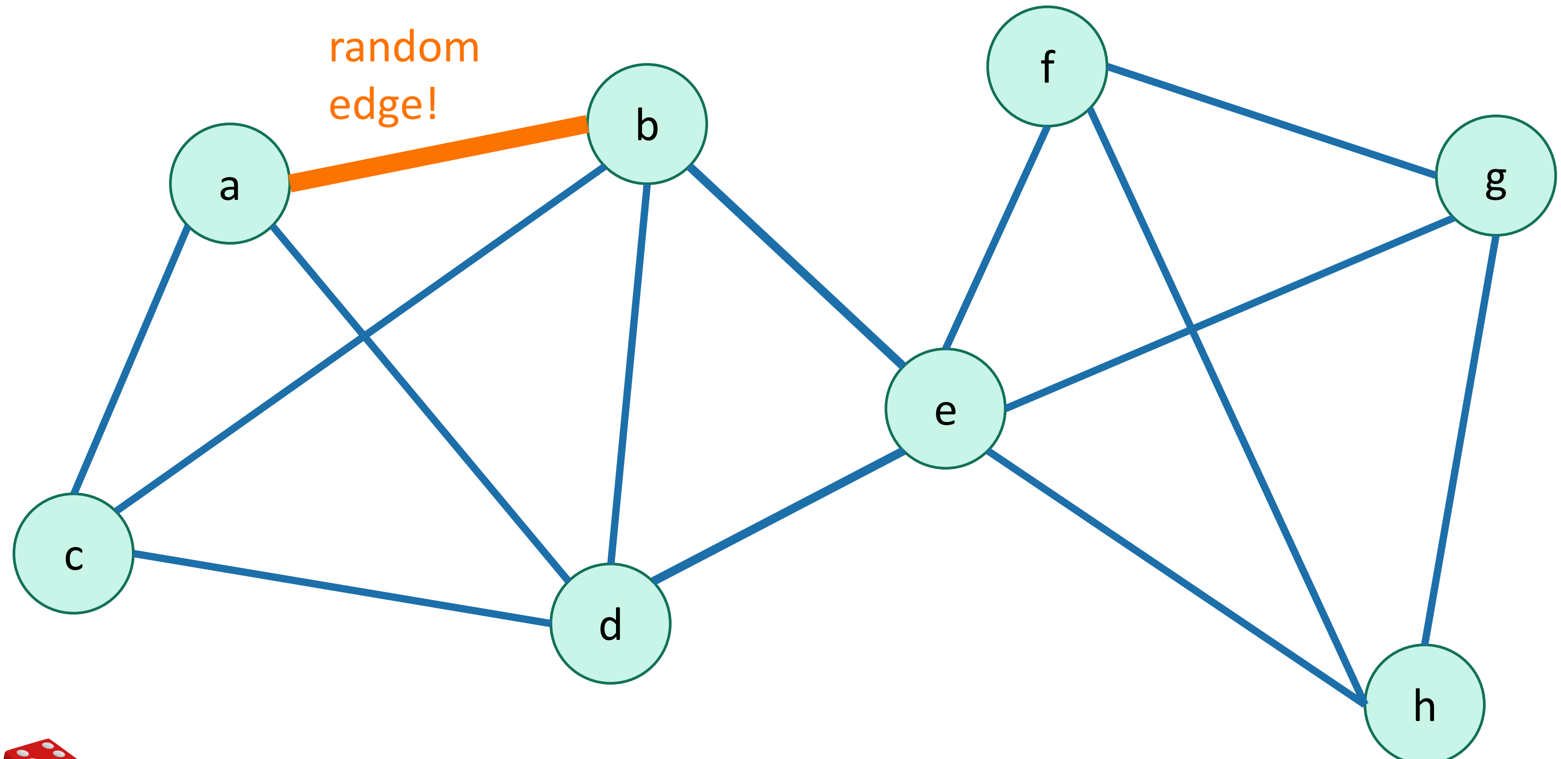


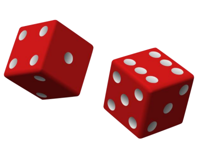
Why is this a good idea? We'll see shortly.

Karger's algorithm

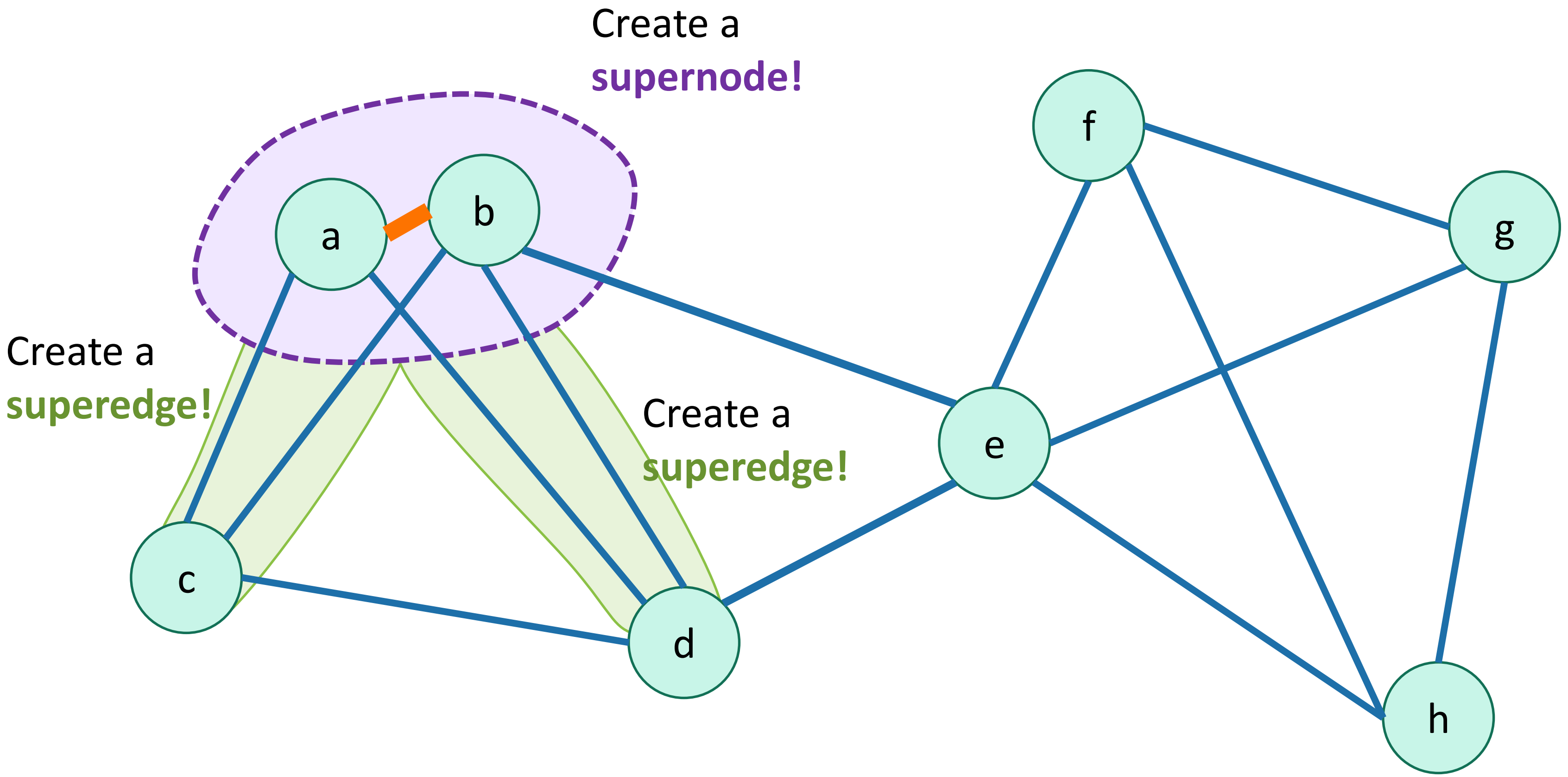


Karger's algorithm

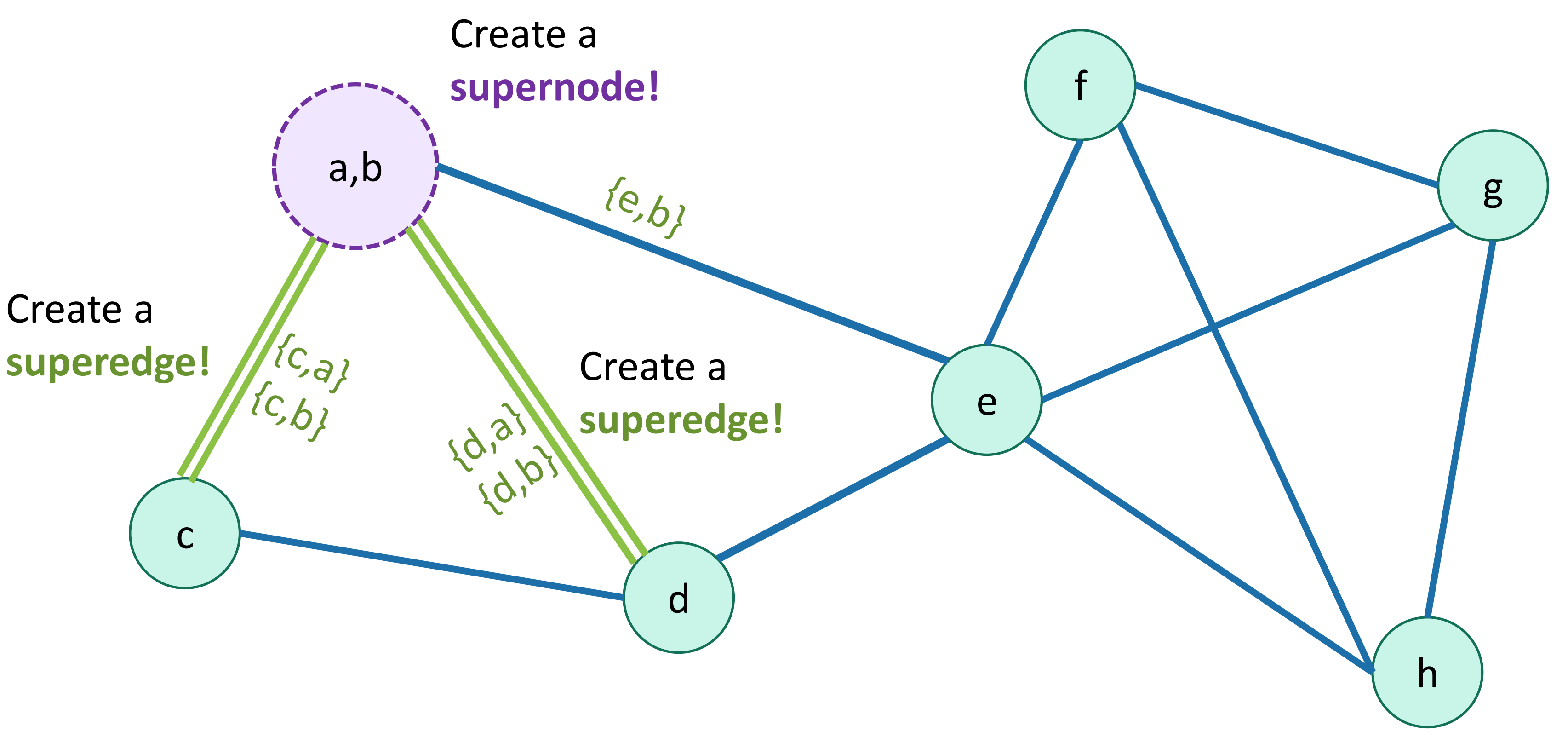




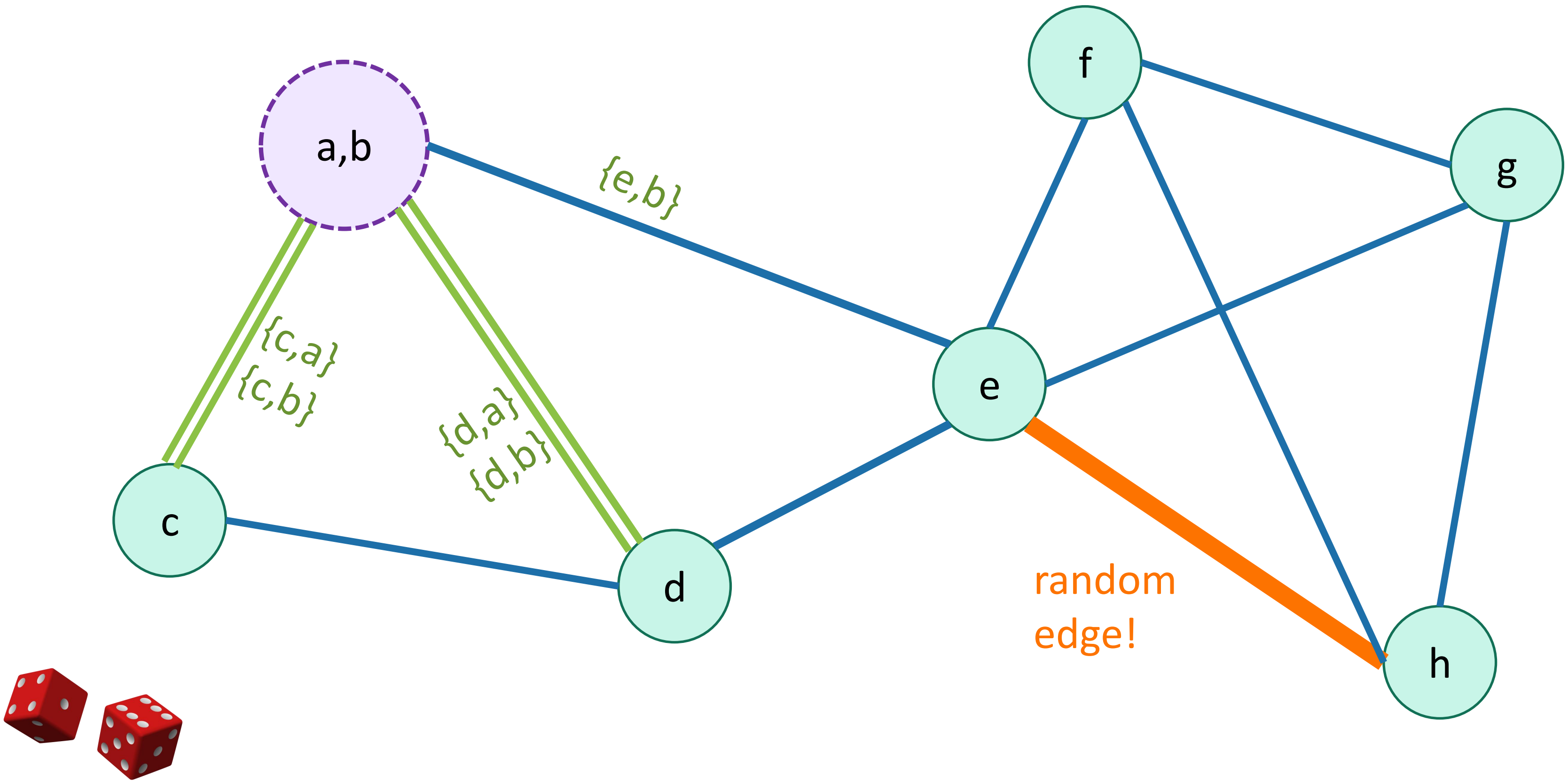
Karger’s algorithm



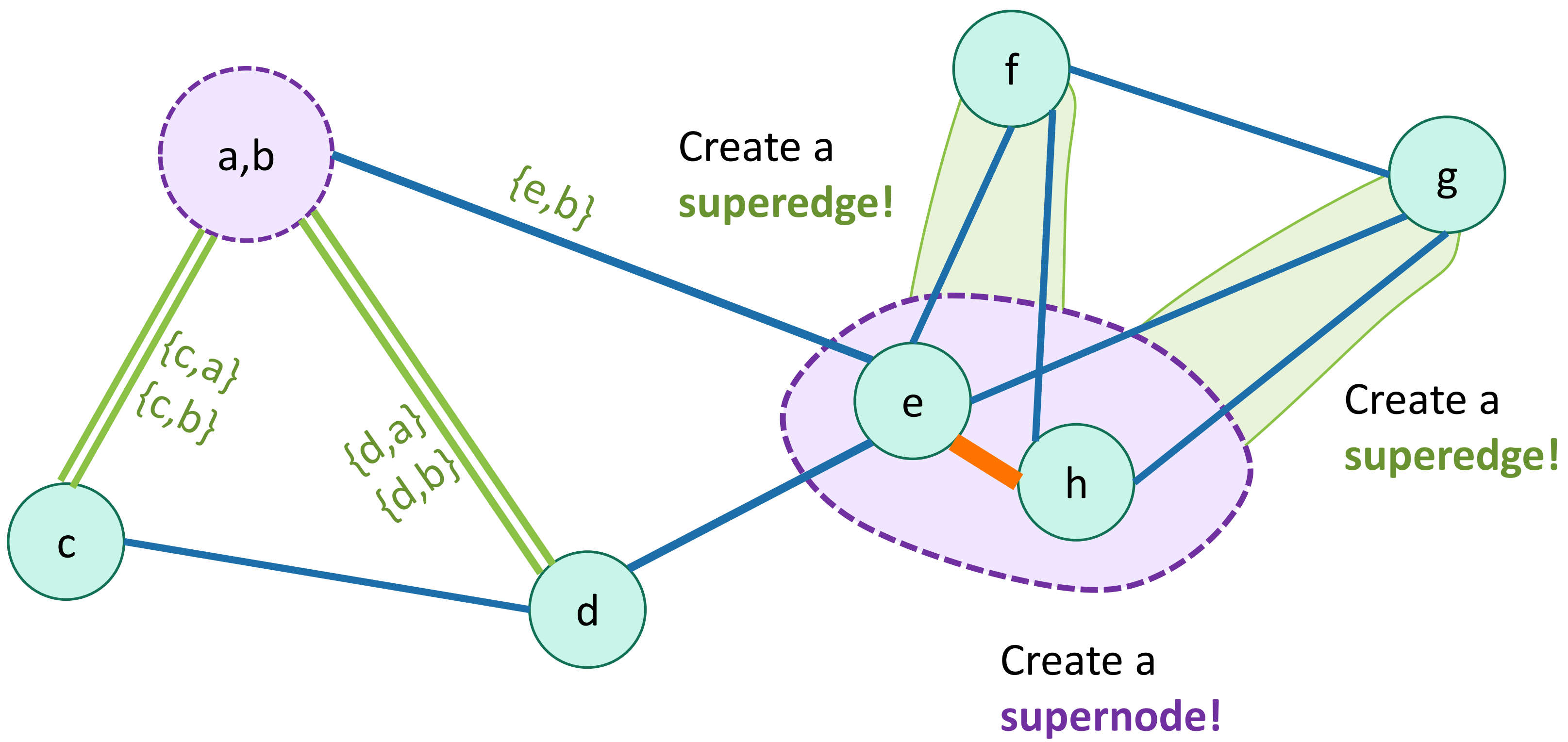
Karger’s algorithm



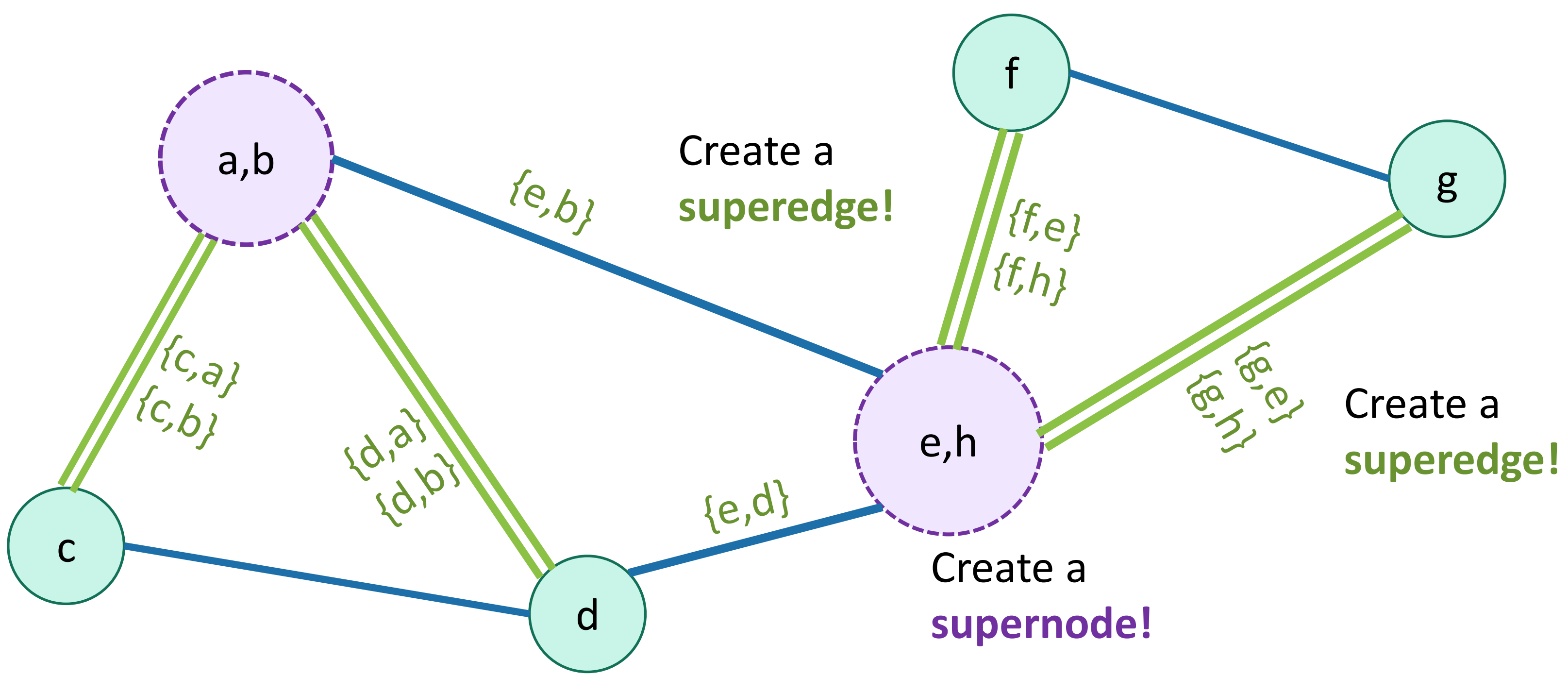
Karger's algorithm



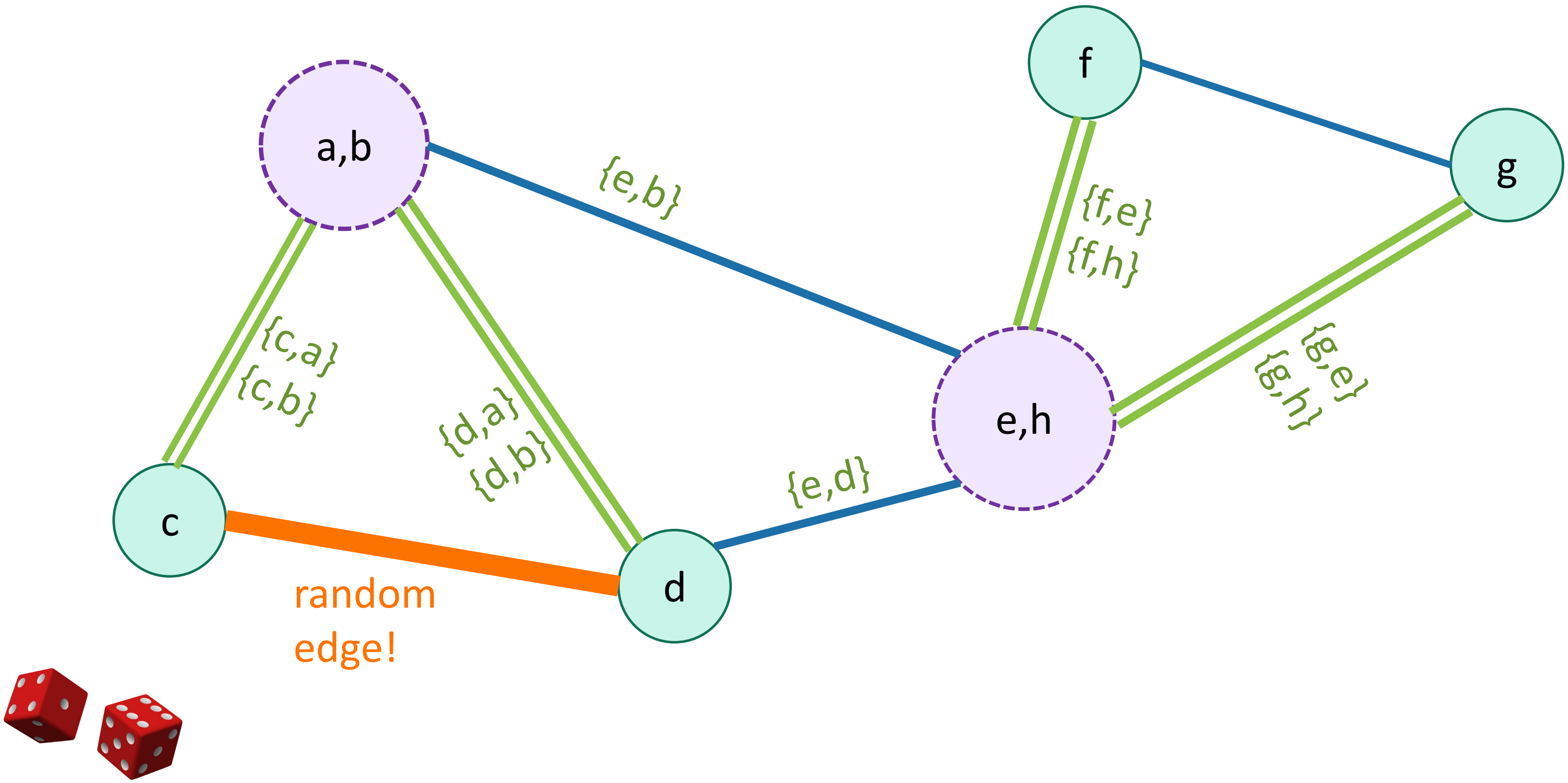
Karger's algorithm



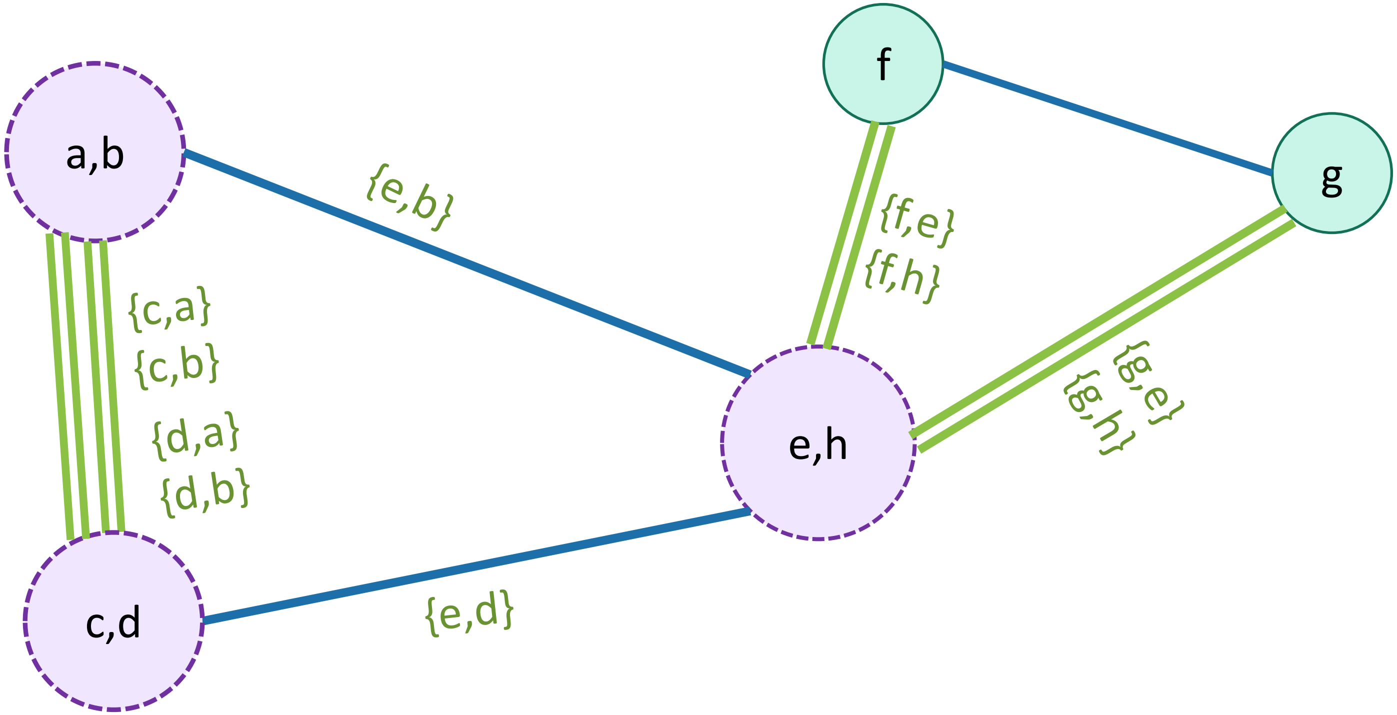
Karger’s algorithm



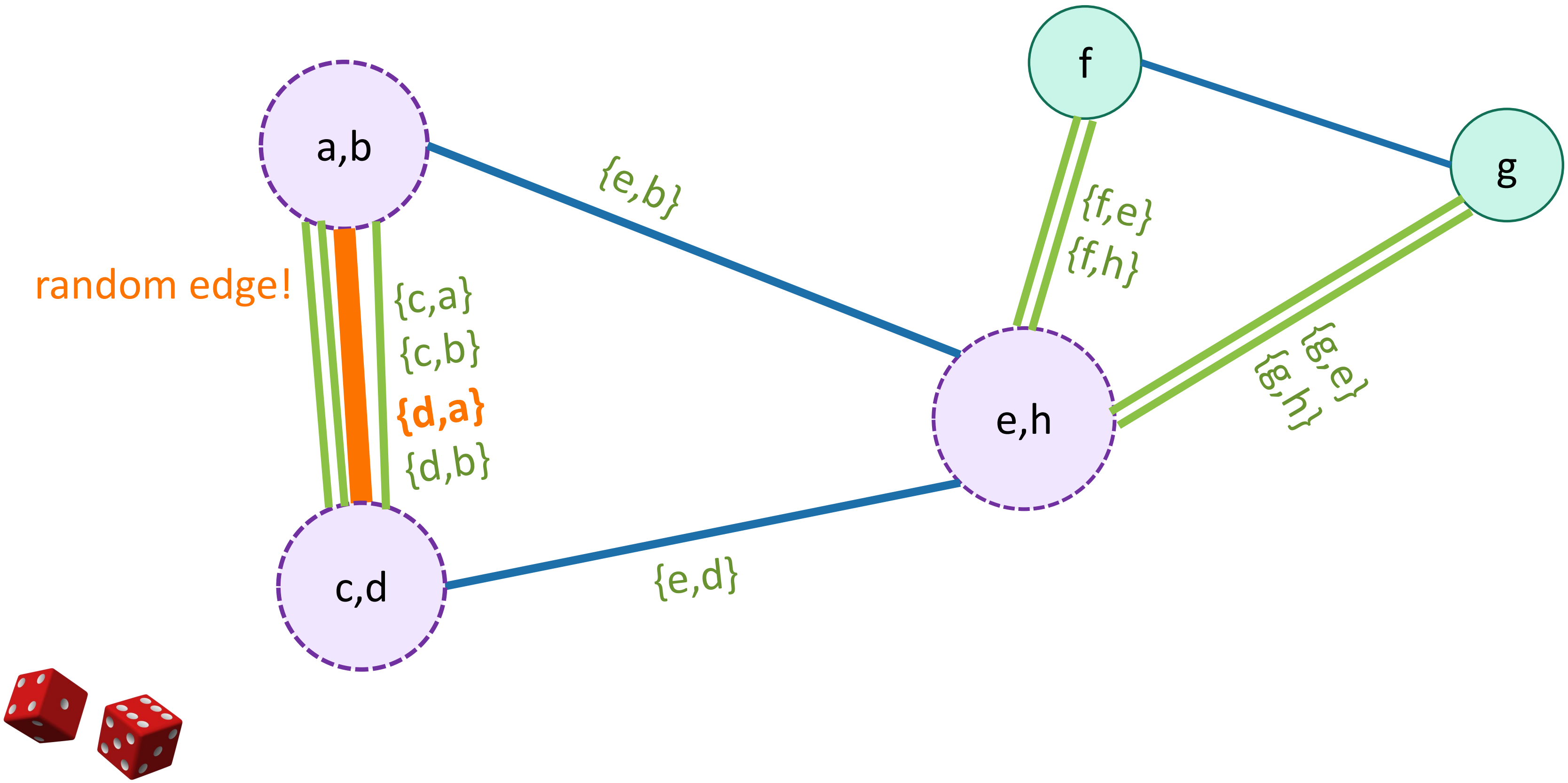
Karger's algorithm



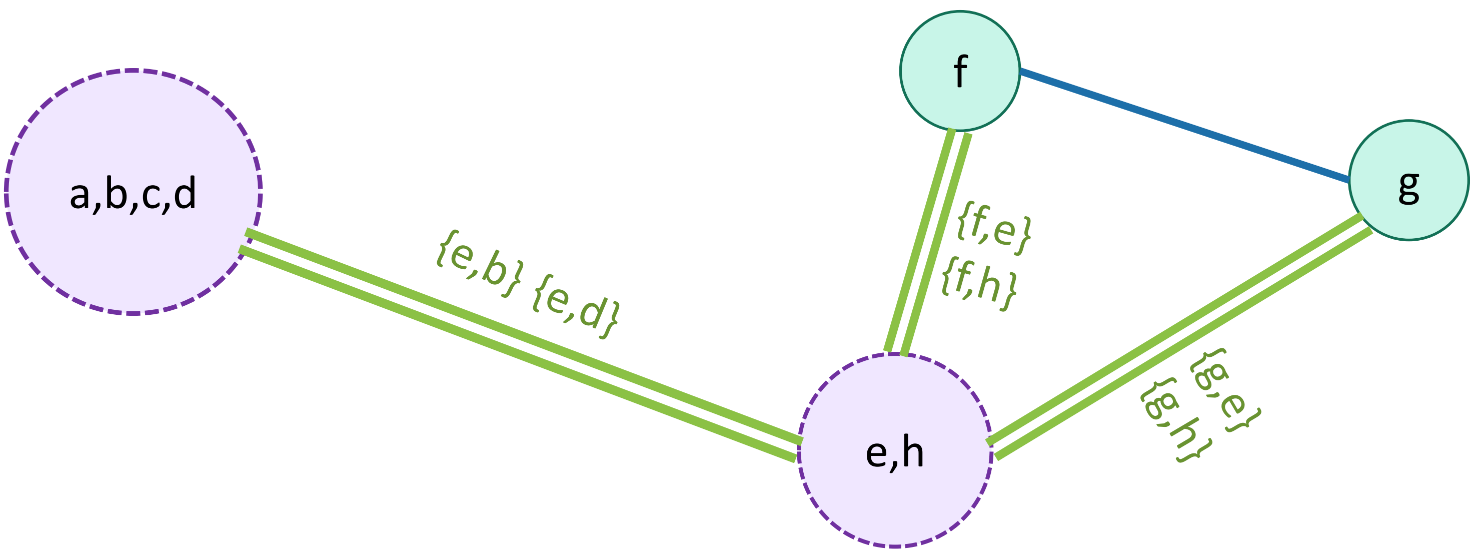
Karger's algorithm



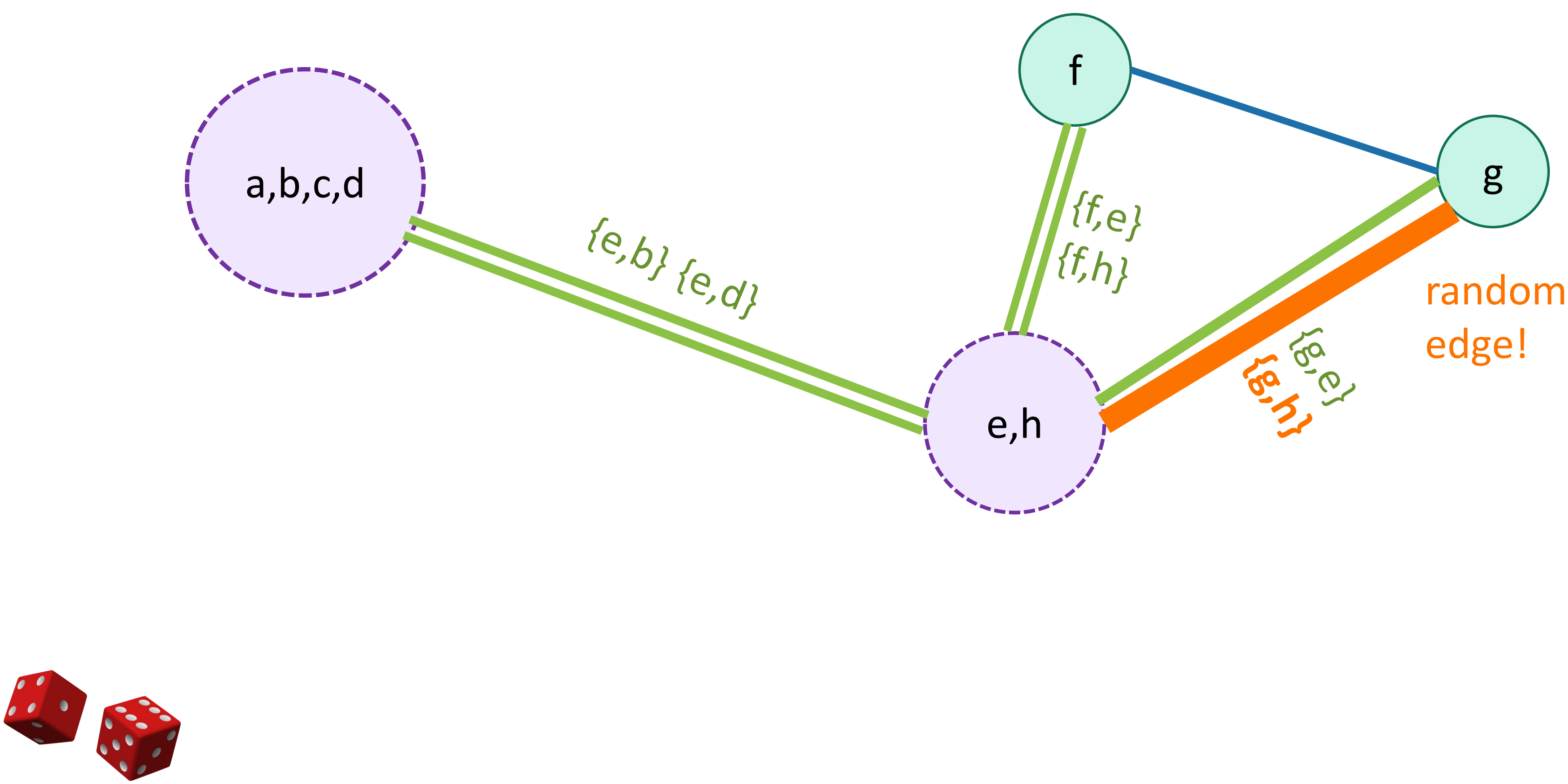
Karger's algorithm



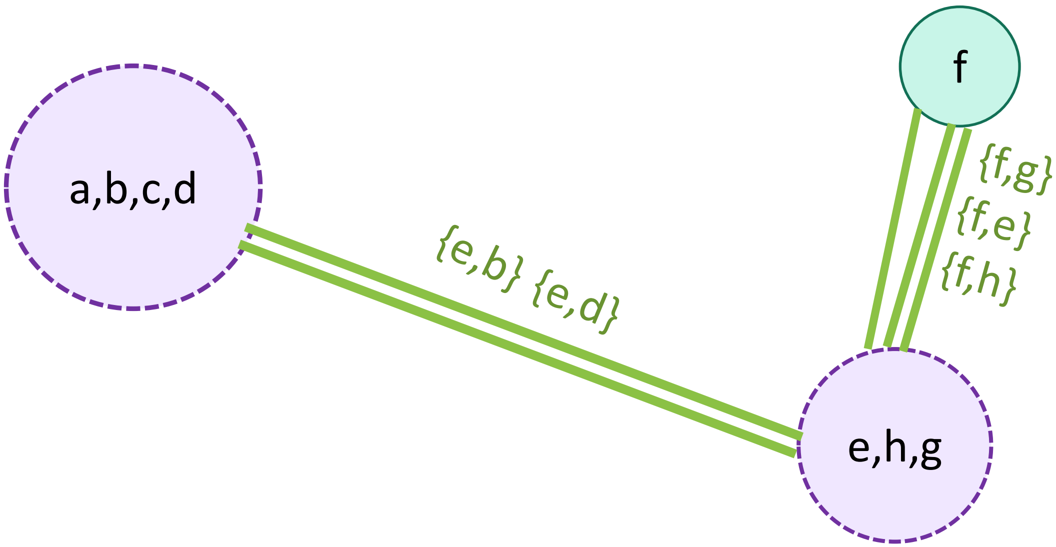
Karger's algorithm



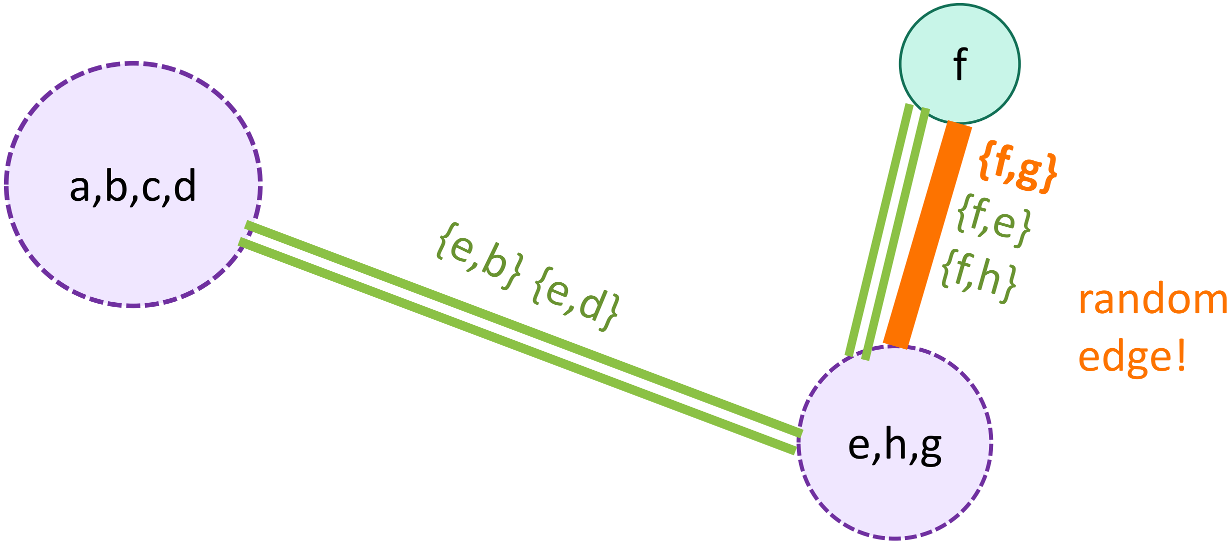
Karger's algorithm

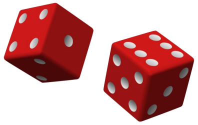


Karger’s algorithm

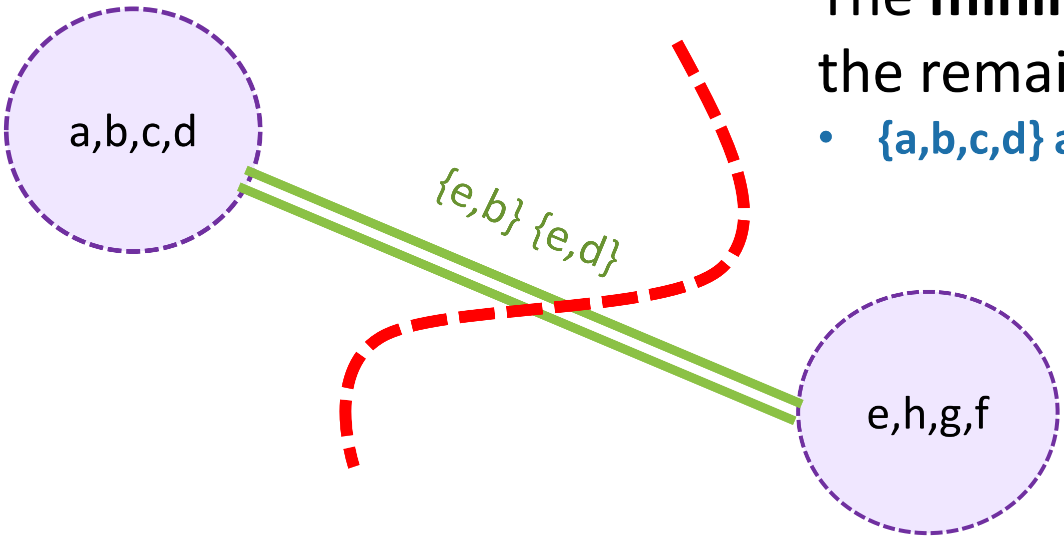


Karger's algorithm





Karger's algorithm



Now stop!

- There are only two nodes left.

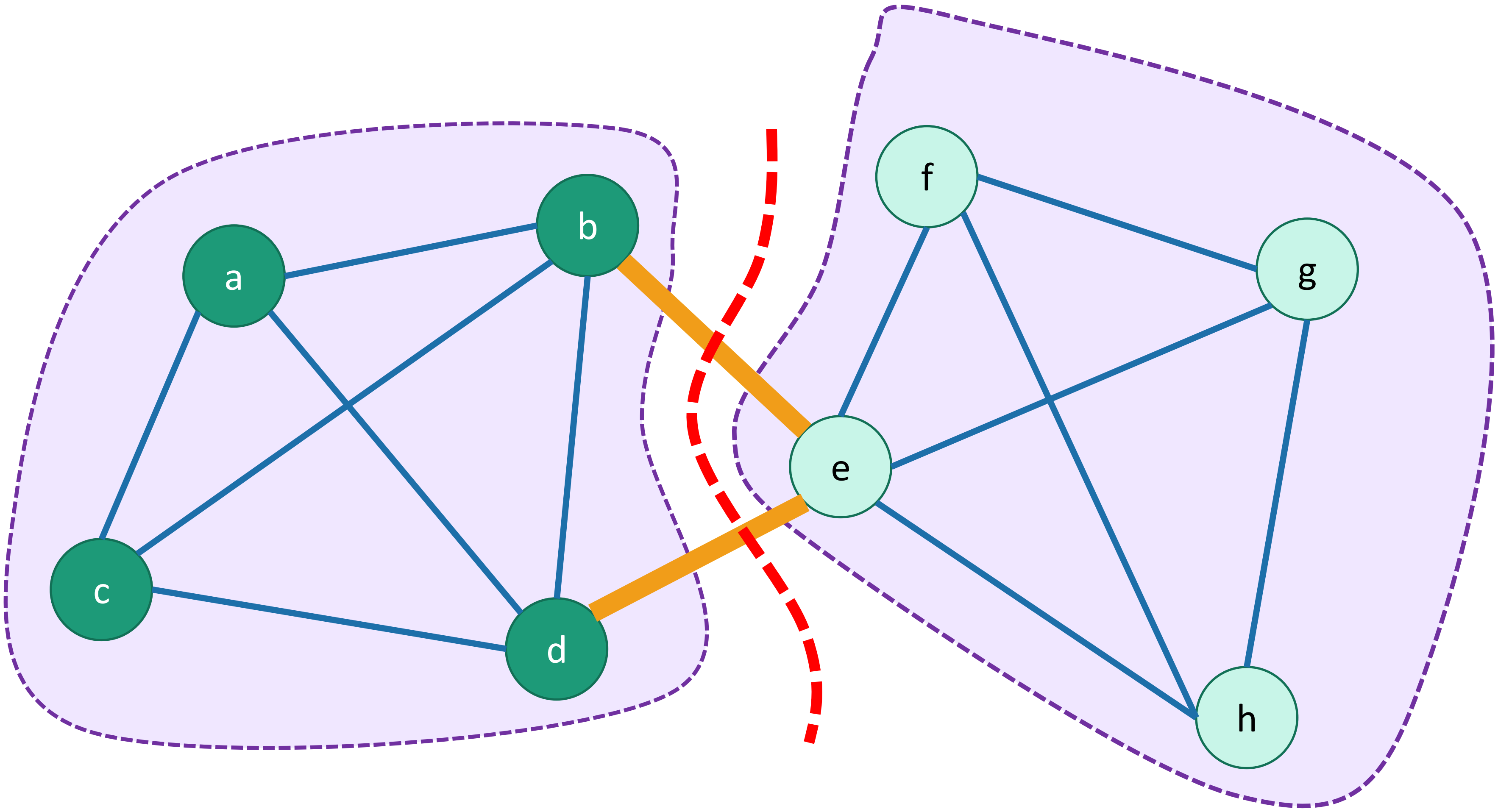
The **minimum cut** is given by the remaining super-nodes:

- **{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?

- Implementation
  - This maintains a secondary “superGraph” which keeps track of superNodes and superEdges
- Running time?
  - We contract at most n-2 edges
    - Each time we contract an edge we get rid of a vertex, and we get rid of at most n – 2 vertices total.
  - Naively each contraction takes time O(n)
    - Maybe there are about n nodes in the superNodes that we are merging.
  - 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 $\overline{u}$ denote the SuperNode in Γ containing u
Say $E_{\overline{u},\overline{v}}$ is the SuperEdge between $\overline{u}, \overline{v}$.

This slide skipped in class

- **Karger**(G=(V,E)):
  - Γ = { SuperNode(v) : v in V } // one supernode for each vertex
  - $E_{\overline{u},\overline{v}}$ = {(u,v)} for (u,v) in E // one superedge for each edge
  - $E_{\overline{u},\overline{v}}$ = {} 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.
    - $F \leftarrow F \setminus E_{\overline{u},\overline{v}}$

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

total runtime O(n^2)

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

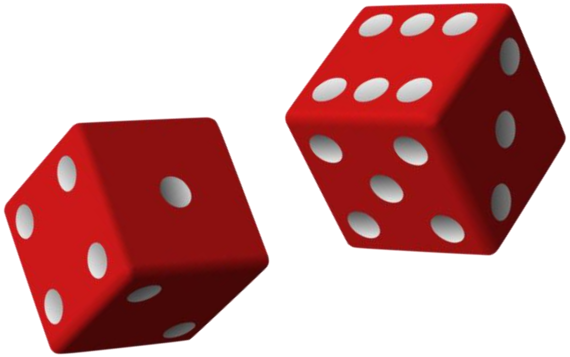
Karger's algorithm

- Does it work?
  - No?

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

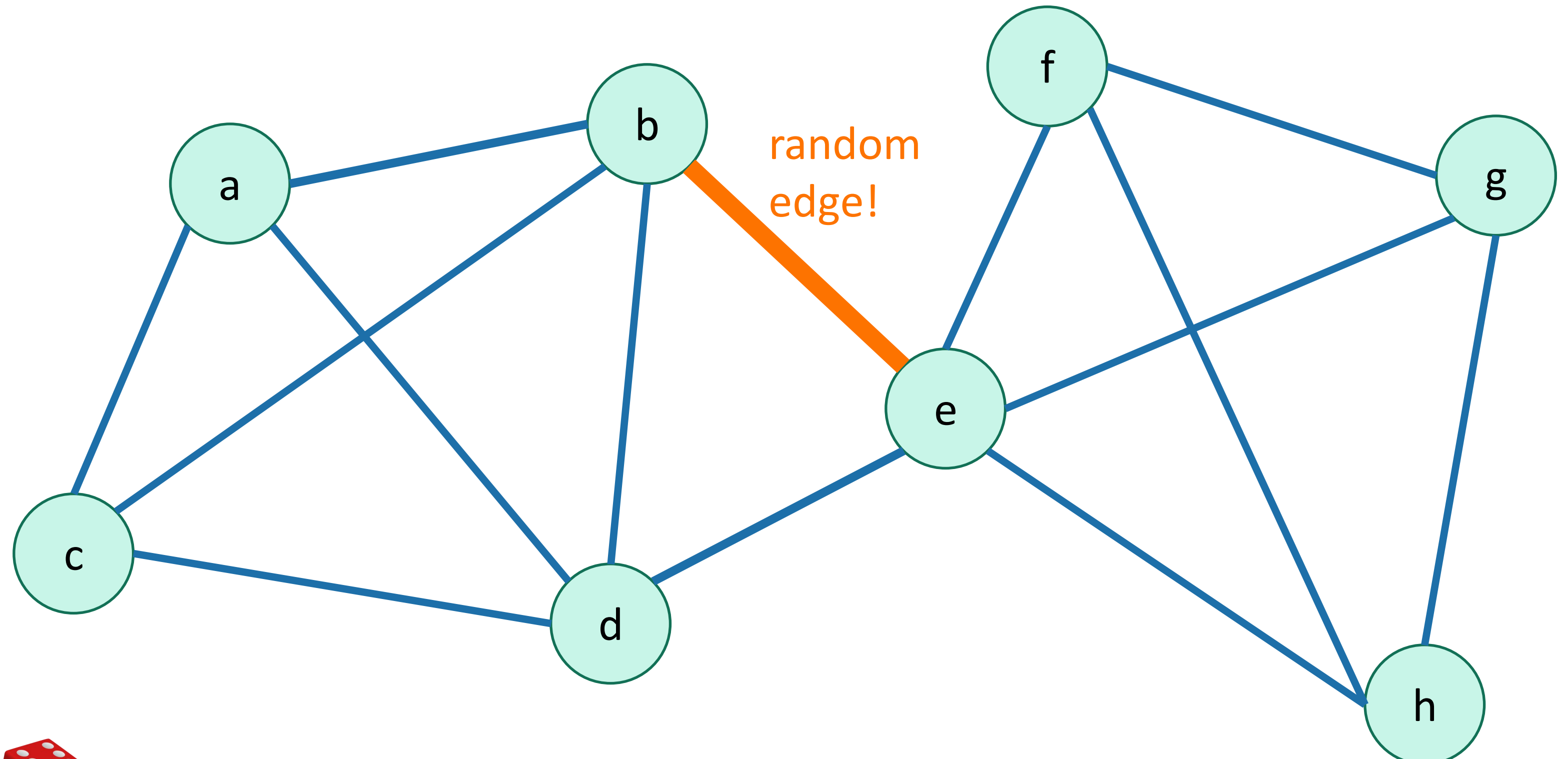
Why did that work?

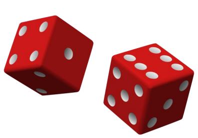
- We got really lucky!
- This could have gone wrong in so many ways.



Karger’s algorithm

Say we had chosen this edge

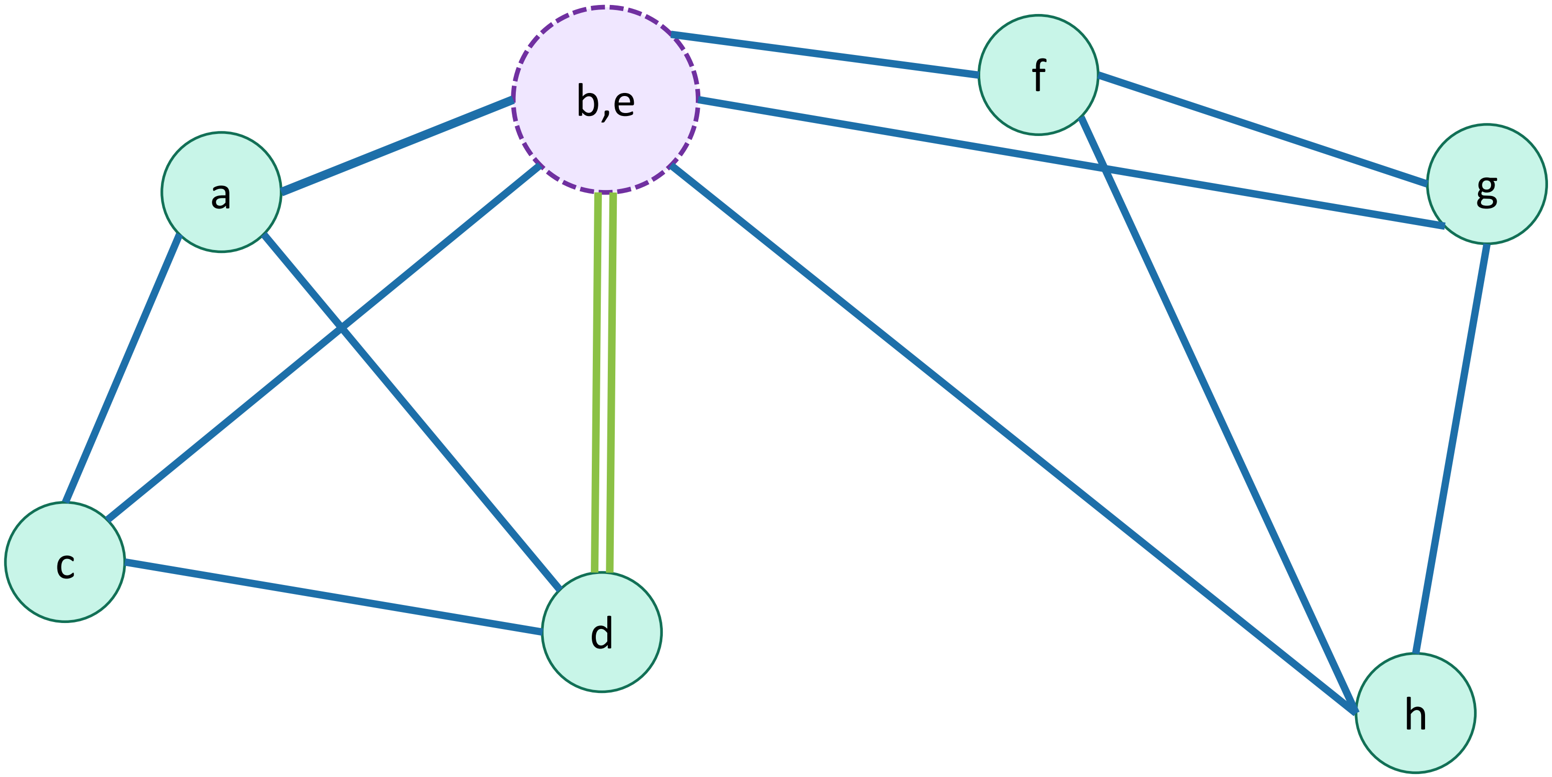




Karger’s algorithm

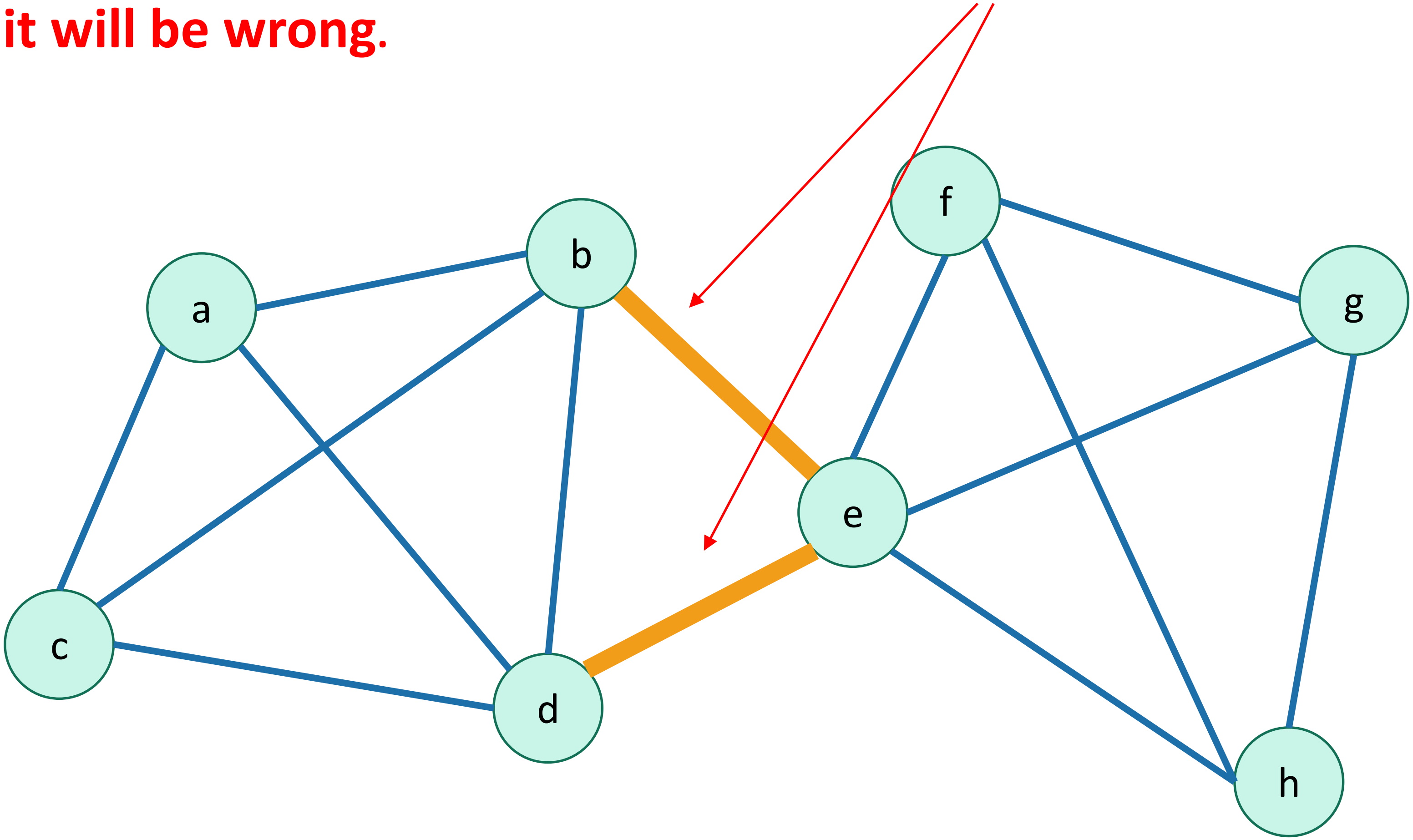
Say we had chosen this edge

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

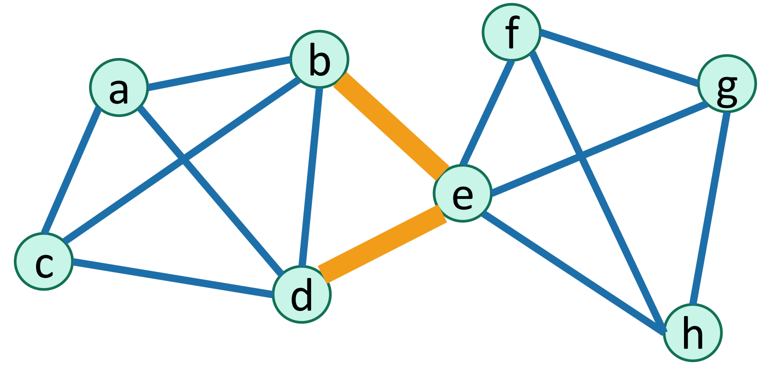


Even worse

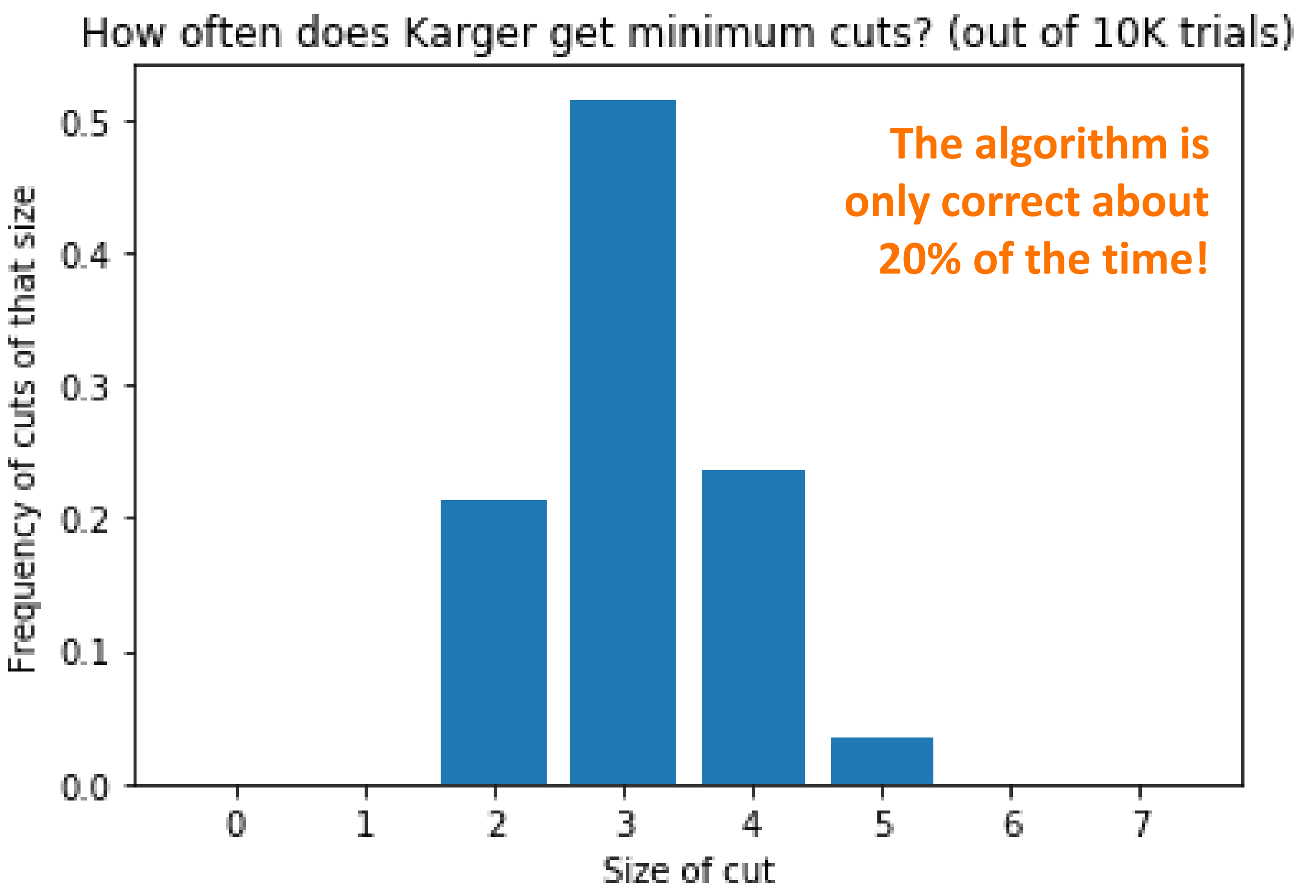
If the algorithm **EVER** chooses either of **these edges**,
it will be wrong.



How likely is that?

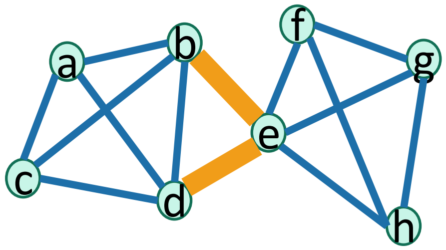


- For this particular graph, I did it 10,000 times:



That doesn’t sound good

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



- Let’s compare Karger’s algorithm to the algorithm:

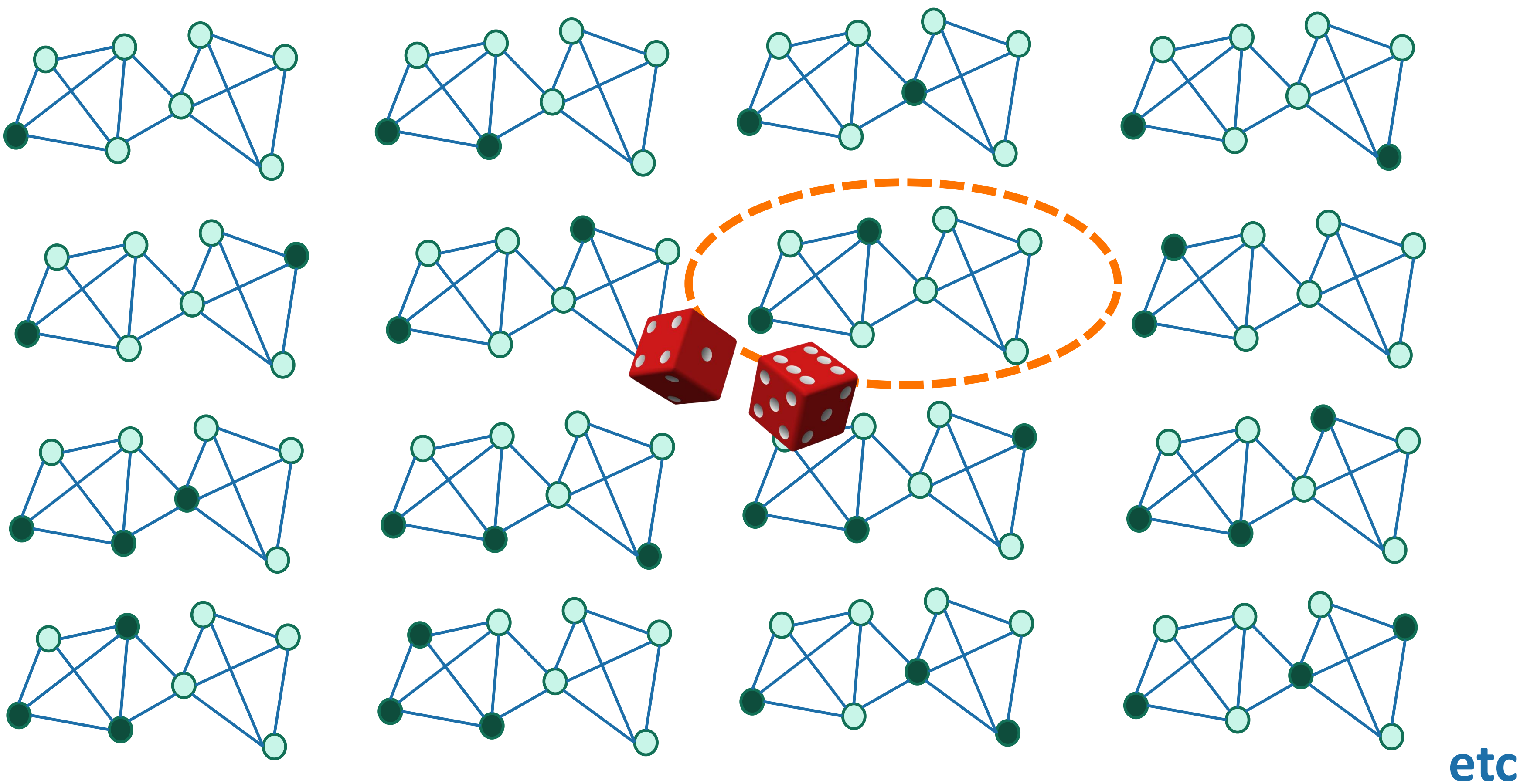
Choose a completely random cut and hope that it’s a minimum cut.

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.

Random cuts

- Suppose that we chose cuts uniformly at random.
  - That is, pick a random way to split the vertices into 2 parts.



Random cuts

- Suppose that we chose cuts uniformly at random.
  - That is, pick a random way to split the vertices into 2 parts.

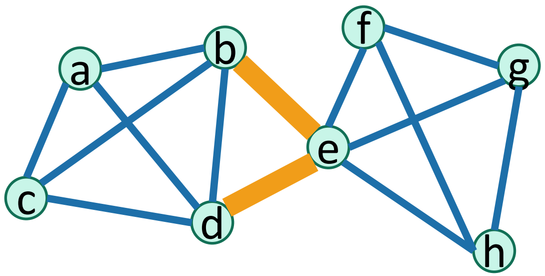
- The probability of choosing the minimum cut is*…

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

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

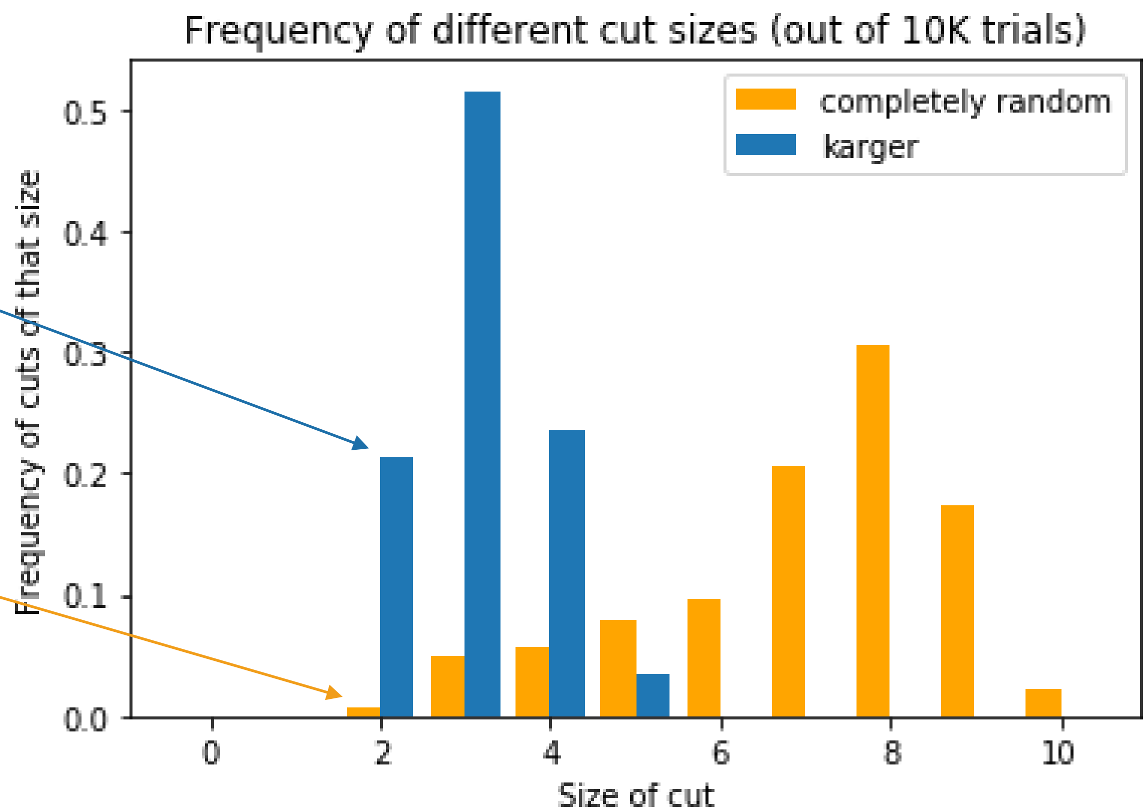
*For this example in particular

Karger is better than completely random!



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

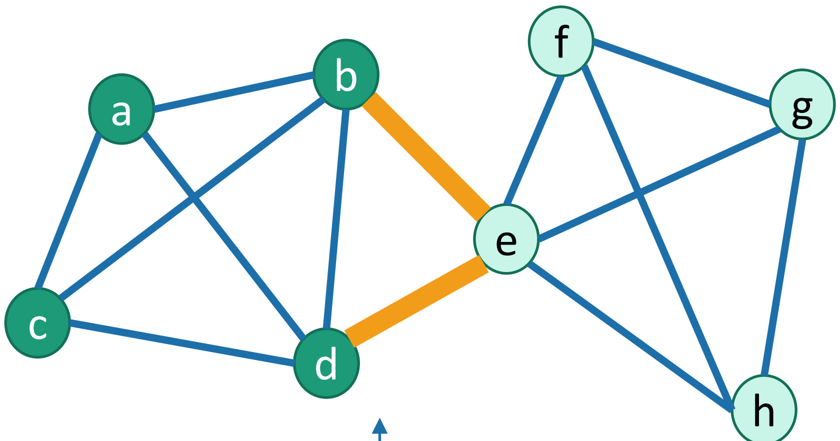
Completely random is correct about 0.8% of the time



What’s going on?

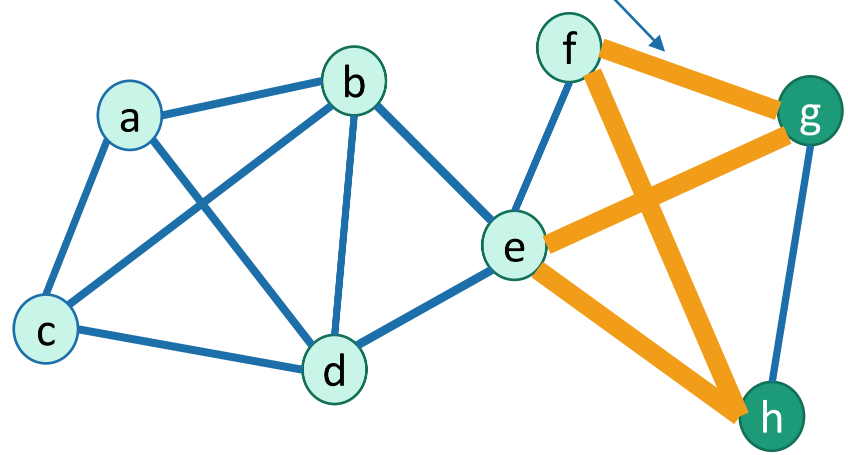
Thing 1: It’s unlikely that Karger will hit the min cut since it’s so small!

- Which is more likely?



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

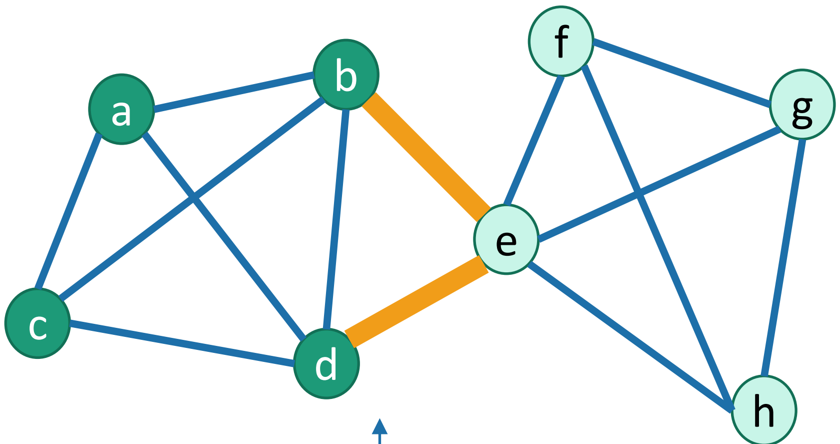
B: The algorithm never chooses any of the edges in **this big cut**.



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

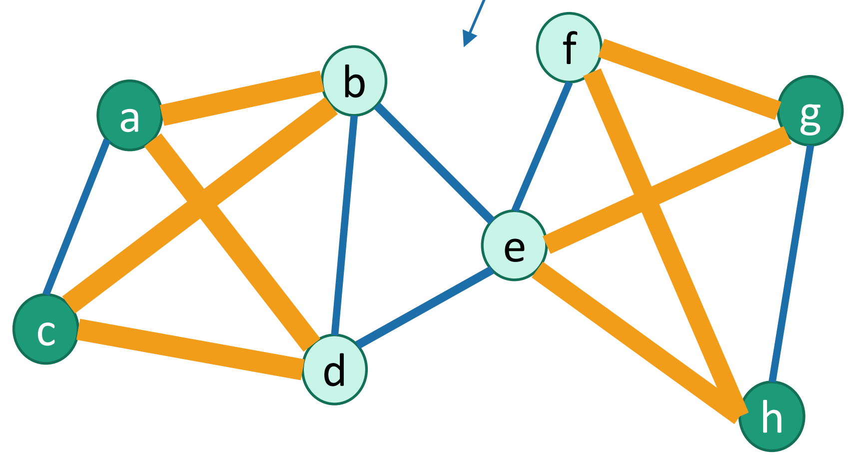
What’s going on?

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



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 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.
- 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 before you see the minimum value?

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

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] = $(1 - 0.2)^{10} \approx 0.1$

In this context

- Run Karger's! The cut size is 6!

- Run Karger's! The cut size is 3!

- Run Karger's! The cut size is 3!

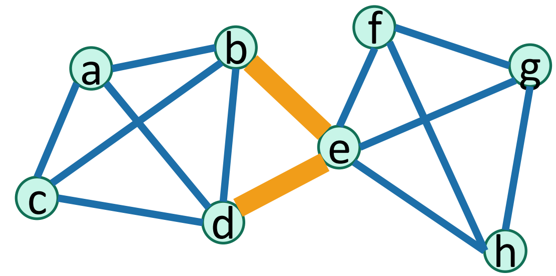
- Run Karger's! The cut size is 2!

Correct!

- Run Karger's! The cut size is 5!

If the success probability is about 20%, then if you run Karger's algorithm 5 times and take the best answer you get, that will likely be correct!

For this particular graph



- Repeat Karger's algorithm about 5 times, and we will get a min cut with decent probability.
  - In contrast, we'd have to choose a random cut about 1/0.008 = 125 times!

Hang on! This "20%" figure just came from running experiments on this particular graph. What about general graphs? Can we prove this?

Also, we should be a bit more precise about this "about 5 times" statement.

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.

Questions

To generalize this approach to all graphs

1. What is the probability that Karger’s algorithm returns a minimum cut?

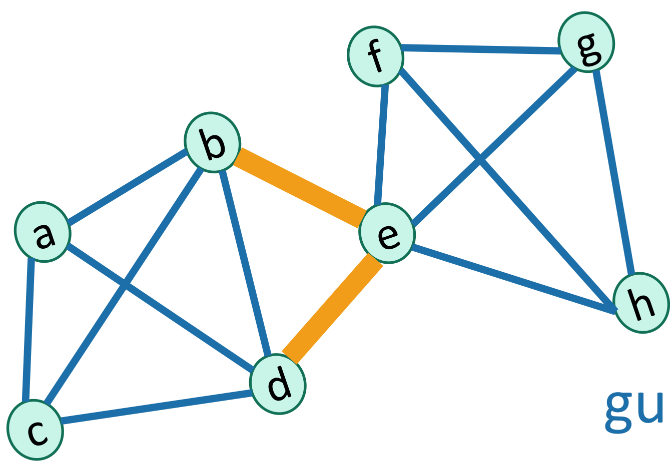
2. How many times should we run Karger’s algorithm to “probably” succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

Answer to Question 1

Claim:

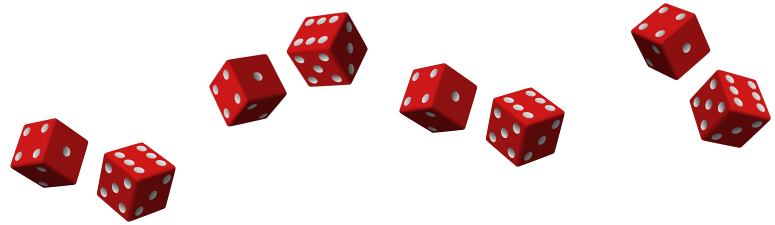
The probability that Karger's algorithm returns a minimum cut is

at least $1/\binom{n}{2}$



In this case, $1/\binom{8}{2} = 0.036$, so we are guaranteed to win at least 3.6% of the time.

Answers



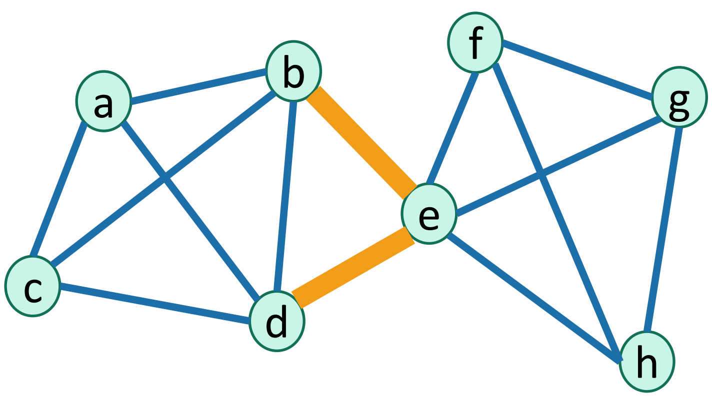
1. What is the probability that Karger's algorithm returns a minimum cut?

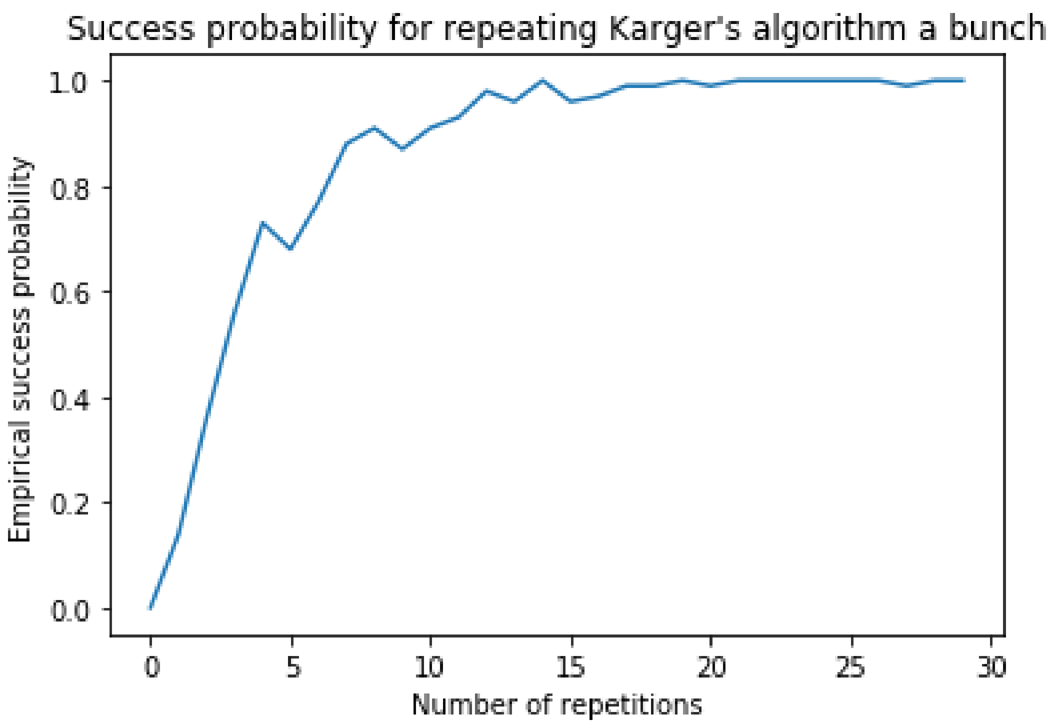
   According to the claim, at most $\frac{1}{\binom{n}{2}}$

2. How many times should we run Karger's algorithm to "probably" succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

Before we prove the Claim

2. How many times should we run Karger's algorithm to succeed with probability $1 - \delta$?





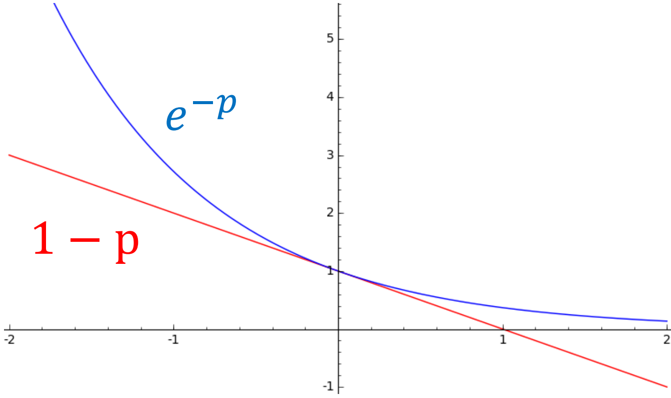
A computation

Punchline: If we repeat **T =** $\binom{n}{2}\ln(1/\delta)$ times, we win with probability at least $\mathbf{1-\delta}$.

- Suppose :
  - the probability of successfully returning a minimum cut is $\boldsymbol{p} \in [\mathbf{0}, \mathbf{1}]$,
  - we want failure probability at most $\delta \in (0,1)$.

Independent

- $\Pr[\text{ don't return a min cut in T trials }] = (1-p)^T$
- So p = $1/\binom{n}{2}$ by the Claim. Let's choose T = $\binom{n}{2}\ln(1/\delta)$.
- $\Pr[\text{ don't return a min cut in T trials }]$
  - $= (1\ -p)^T$
  - $\leq (e^{-p})^T$
  - $= e^{-pT}$
  - $= e^{-\ln\left(\frac{1}{\delta}\right)}$
  - $= \delta$



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

Theorem

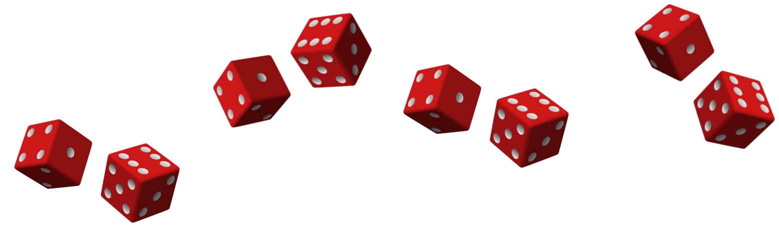
Assuming the claim about $1/\binom{n}{2}$ …

- Suppose G has n vertices.

- Consider the following algorithm:
  - bestCut = None
  - **for** $\mathrm{t} = 1, \dots, \binom{n}{2} \ln\left(\frac{1}{\delta}\right)$:
    - candidateCut ← **Karger**(G)
    - **if** candidateCut is smaller than bestCut:
      - bestCut ← candidateCut
  - **return** bestCut

- Then $\Pr[\text{ this doesn't return a min cut }] \le \delta.$

Answers



1. What is the probability that Karger's algorithm returns a minimum cut?

   According to the claim, at most $\frac{1}{\binom{n}{2}}$

2. How many times should we run Karger's algorithm to "probably" succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

   $\binom{n}{2} \log\left(\frac{1}{\delta}\right)$ times.

What's the running time?

- $\binom{n}{2} \ln\left(\frac{1}{\delta}\right)$ repetitions, and O(n^2) per repetition.
- So, $O\left(n^2 \cdot \binom{n}{2} \ln\left(\frac{1}{\delta}\right)\right) = \mathrm{O}(\mathrm{n}^4)$

Treating δ as constant.

Again we can do better with a union-find data structure. Write pseudocode for—or better yet, implement—a fast version of Karger's algorithm! How fast can you make the asymptotic running time?

Theorem

Assuming the claim about $1/\binom{n}{2}$ …

Suppose G has n vertices. Then [repeating Karger's algorithm] finds a min cut in G with probability at least 0.99 in time $O(n^4)$.

Now let's prove the claim…

Claim

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

at least $^{1}/_{\binom{n}{2}}$

Now let's prove that claim

Say that S* is a minimum cut.

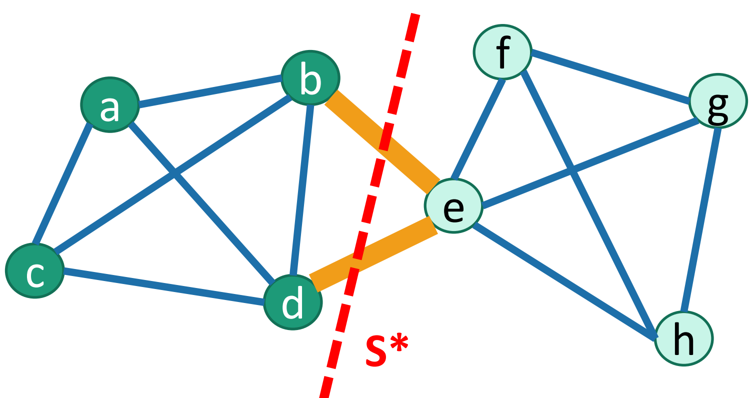
- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- **PR**[return S*] = **PR**[none of the e_i cross S*]

$= \textbf{PR}[\ e_1 \text{ doesn't cross } S^*\ ]$

$\times\ \textbf{PR}[\ e_2 \text{ doesn't cross } S^* \mid e_1 \text{ doesn't cross } S^*\ ]$

…

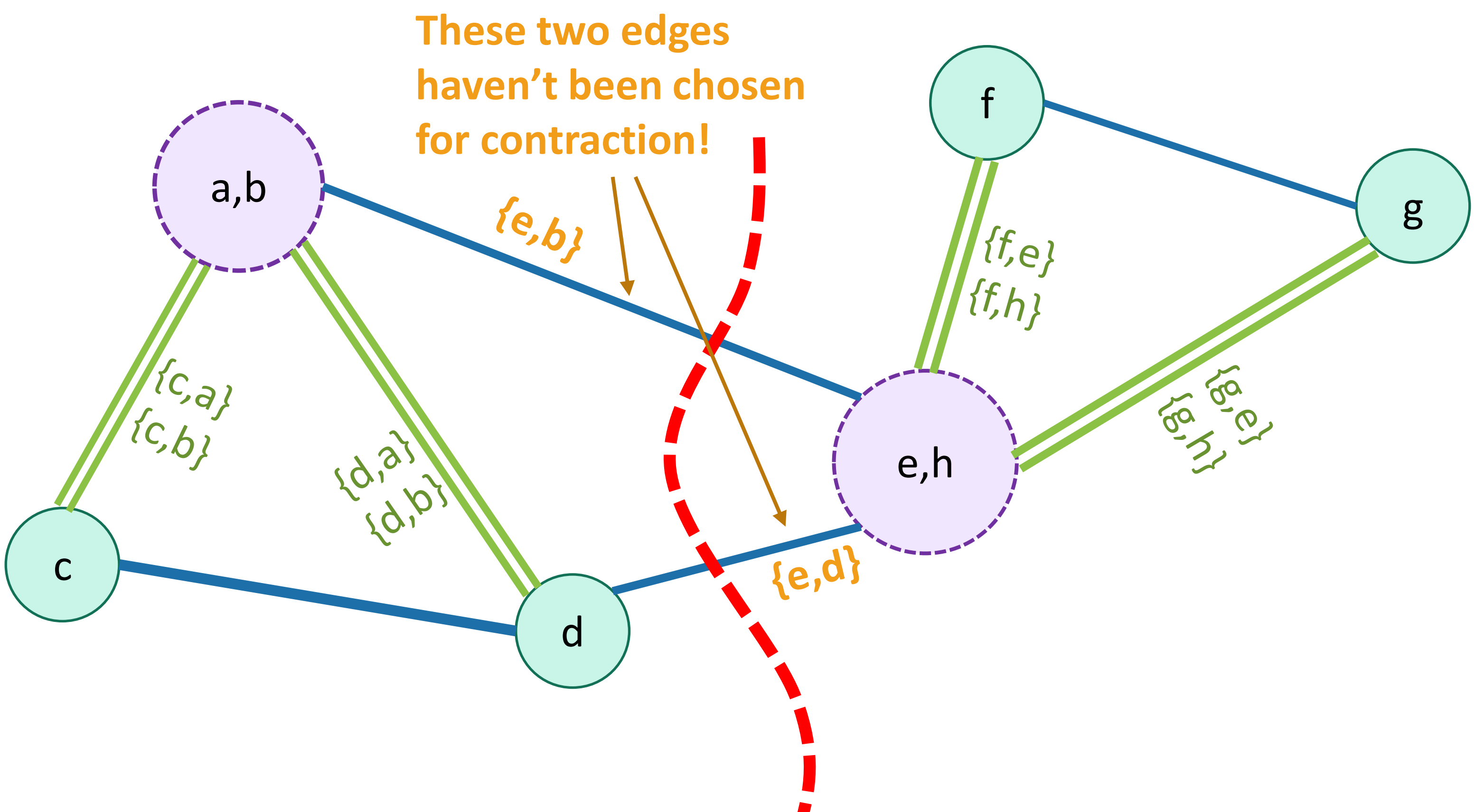
$\times\ \textbf{PR}[\ e_{n-2} \text{ doesn't cross } S^* \mid e_1,\ldots,e_{n-3} \text{ don't cross } S^*\ ]$



Focus in on:

PR[e_j doesn't cross S* | $e_1,\ldots,e_{j-1}$ don't cross S*]

- Suppose: After j-1 iterations, we haven't messed up yet!
- What's the probability of messing up now?



Focus in on:

PR[e_j doesn't cross S* | $e_1,\ldots,e_{j-1}$ don't cross S*]

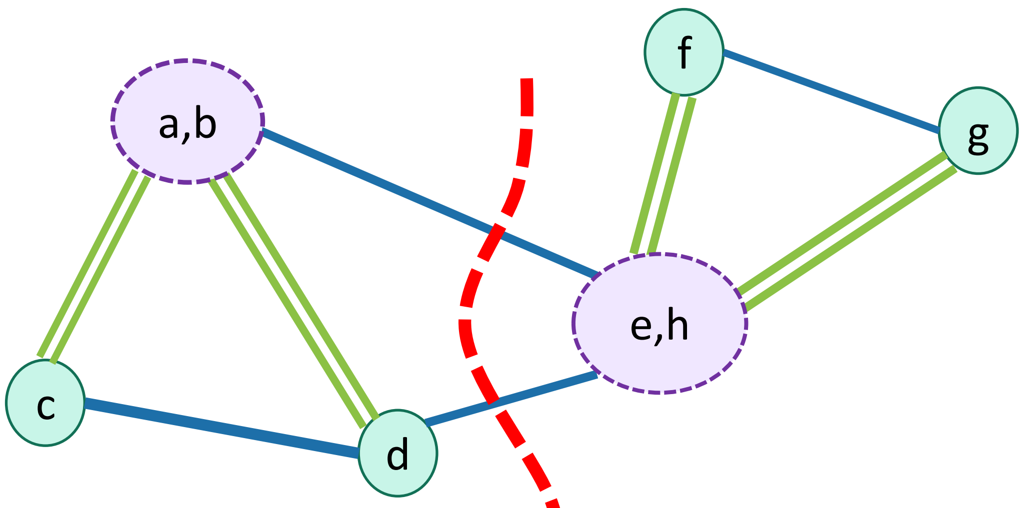
- Suppose: After j-1 iterations, we haven't messed up yet!
- What's the probability of messing up now?

- Say there are k edges that cross S*
- Every remaining node has degree at least k.
  - Otherwise we'd have a smaller cut.
- Thus, there are at least (n-j+1)k/2 edges total.
  - b/c there are n - j + 1 nodes left, each with degree at least k.

Recall: the **degree** of the vertex is the number of edges coming out of it.

So the probability that we choose one of the k edges crossing S* at step j is at most:

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



Focus in on:

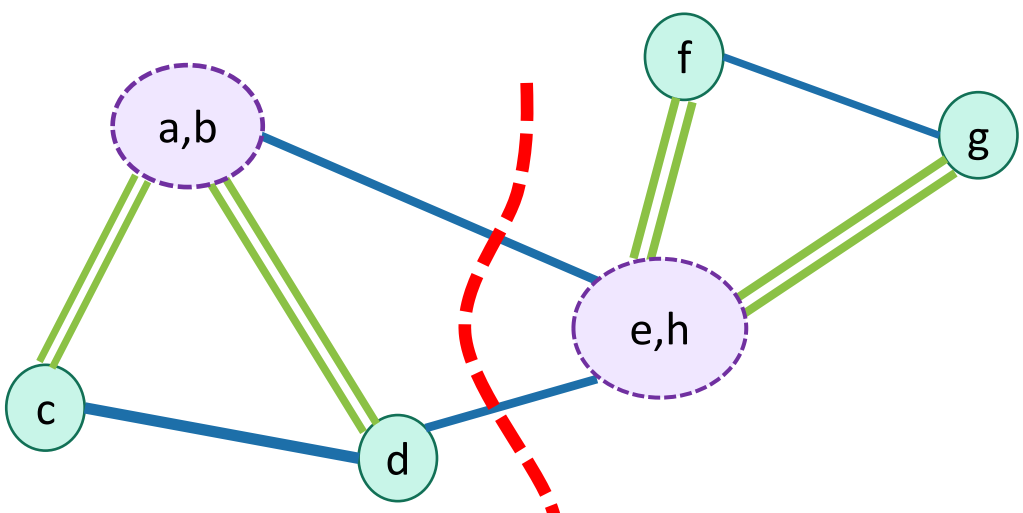
PR[e_j doesn't cross S* | $e_1,\ldots,e_{j-1}$ don't cross S*]

- So the probability that we choose one of the k edges crossing S* at step j is at most:

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

- The probability we **don't** choose one of the k edges is at least:

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



Now let's prove that claim

Say that S* is a minimum cut.

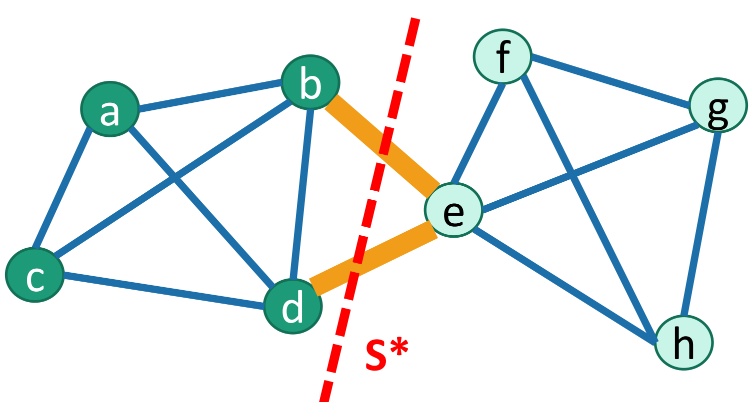
- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- **PR**[return S*] = **PR**[none of the e_i cross S*]

  = **PR**[e_1 doesn't cross S*]

  × **PR**[e_2 doesn't cross S* | e_1 doesn't cross S*]

  …

  × **PR**[e_{n-2} doesn't cross S* | $e_1, \ldots, e_{n-3}$ don't cross S*]

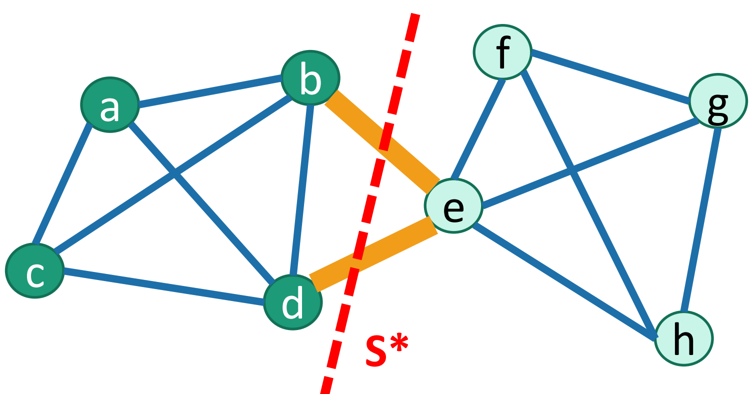


Now let's prove that claim

Say that S* is a minimum cut.

- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- **PR**[return S*] = **PR**[none of the e_i cross S*]

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



Now let's prove that claim

Say that S* is a minimum cut.

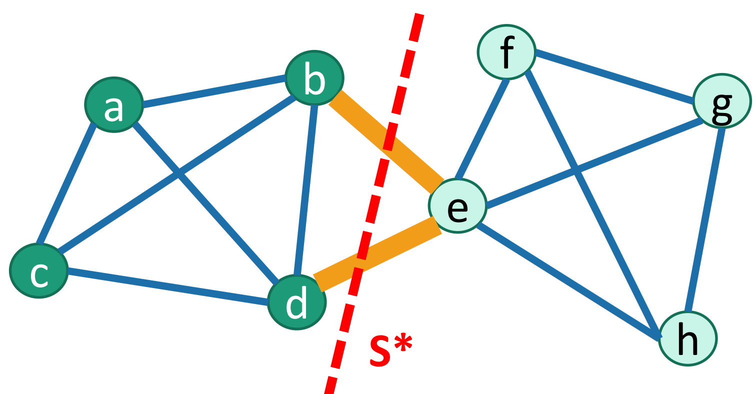
- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- **PR**[return S*] = **PR**[none of the e_i cross S*]

$$= \left(\frac{\cancel{n-2}}{n}\right)\left(\frac{\cancel{n-3}}{n-1}\right)\left(\frac{\cancel{n-4}}{\cancel{n-2}}\right)\left(\frac{\cancel{n-5}}{\cancel{n-3}}\right)\left(\frac{\cancel{n-6}}{\cancel{n-4}}\right)\cdots\left(\frac{\cancel{4}}{\cancel{6}}\right)\left(\frac{\cancel{3}}{\cancel{5}}\right)\left(\frac{2}{\cancel{4}}\right)\left(\frac{1}{\cancel{3}}\right)$$

$$= \left(\frac{2}{n(n-1)}\right)$$

$$= \frac{1}{\binom{n}{2}}$$

CLAIM PROVED



Theorem

Assuming the claim about $1/\binom{n}{2}$ …

Suppose G has n vertices. Then [repeating Karger's algorithm] finds a min cut in G with probability at least 0.99 in time $O(n^4)$.

That proves this Theorem!

What have we learned?

- If we randomly contract edges:
  - It's unlikely that we'll end up with a min cut.
  - But it's not **TOO** unlikely
  - By repeating, we likely will find a min cut.

Here I chose $\delta = 0.01$ just for concreteness.

- Repeating this process:
  - Finds a **global min cut in time O(n^4), with probability 0.99**.
  - We can run a bit faster if we use a **union-find** data structure.

*Note, in the lecture notes, we take $\delta = \frac{1}{n}$, which makes the running time O(n^4log(n)). It depends on how sure you want to be!

More generally

- Whenever we have a Monte-Carlo algorithm with a small success probability, we can **boost** the success probability by repeating it a bunch and taking the best solution.

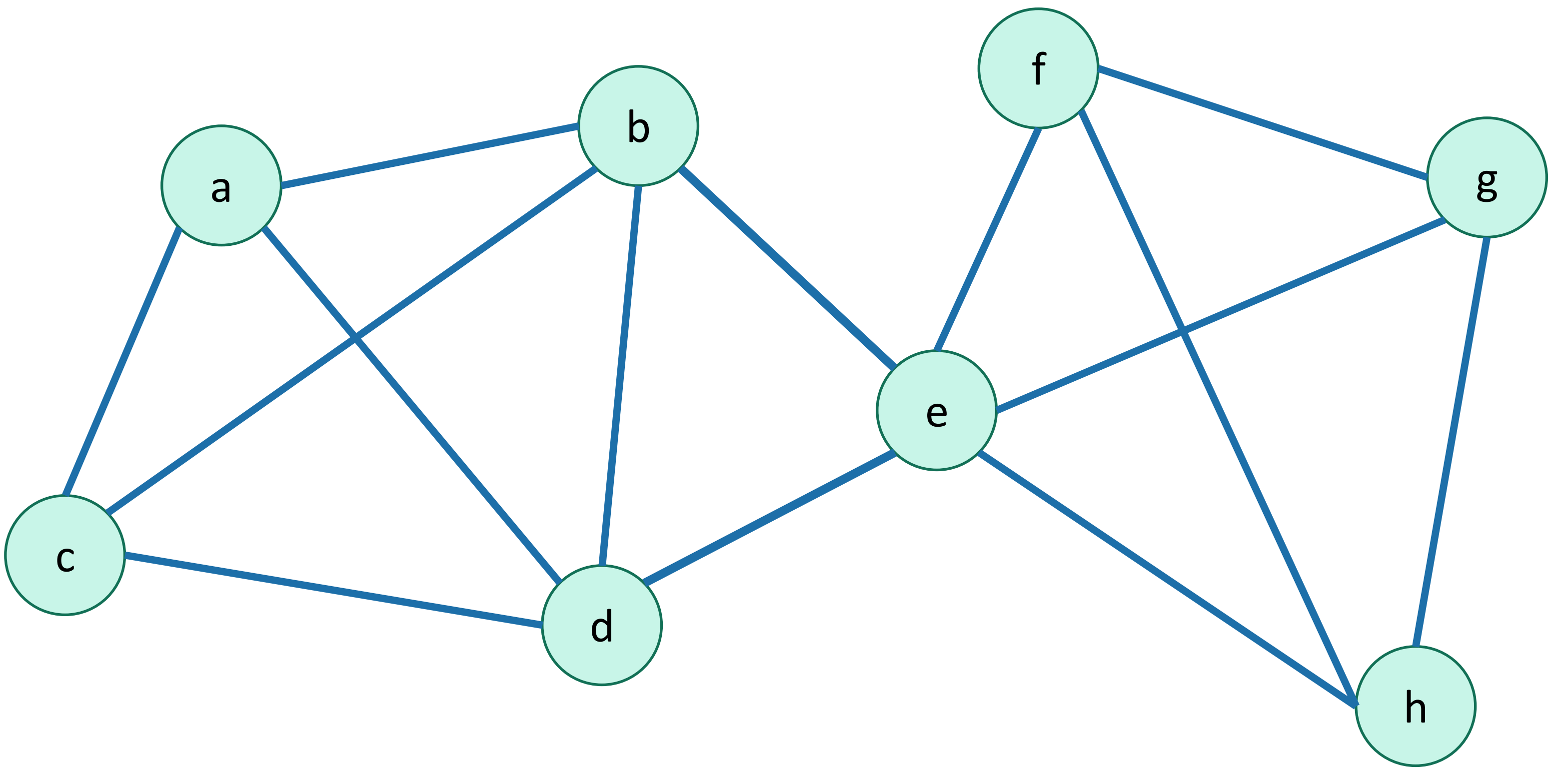
Can we do better?

- Repeating $O(n^2)$ times is pretty expensive.
  - $O(n^4)$ total runtime to get success probability 0.99.

- The **Karger-Stein Algorithm** will do better!
  - The trick is that we'll do the repetitions in a clever way.
  - $O(n^2\log^2(n))$ runtime for the same success probability.
  - **Warning!** This is a tricky algorithm! We'll sketch the approach here: the important part is the high-level idea, not the details of the computations.

To see how we might save on repetitions, let's run through Karger's algorithm again.

Karger's algorithm

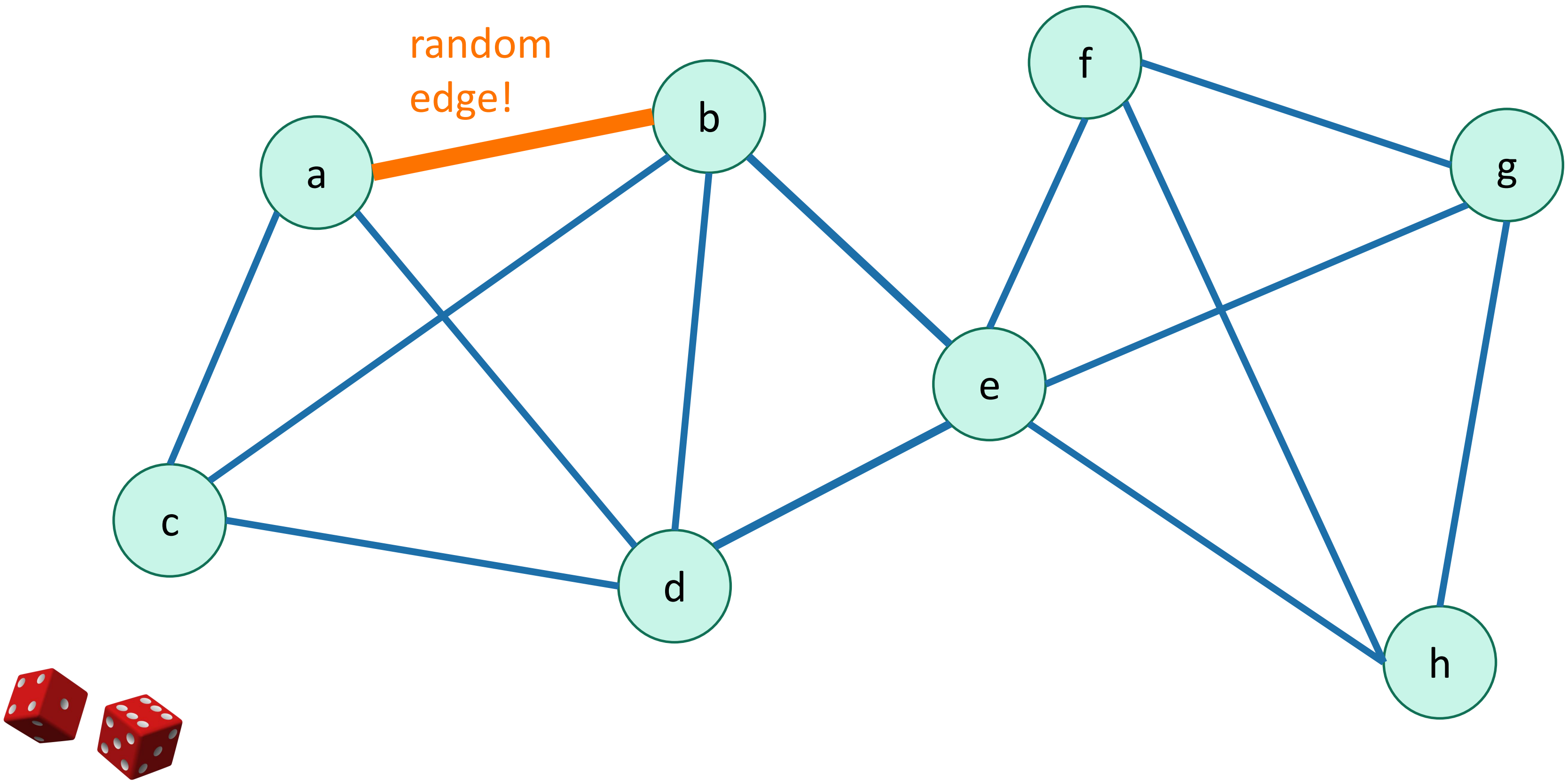


Karger's algorithm

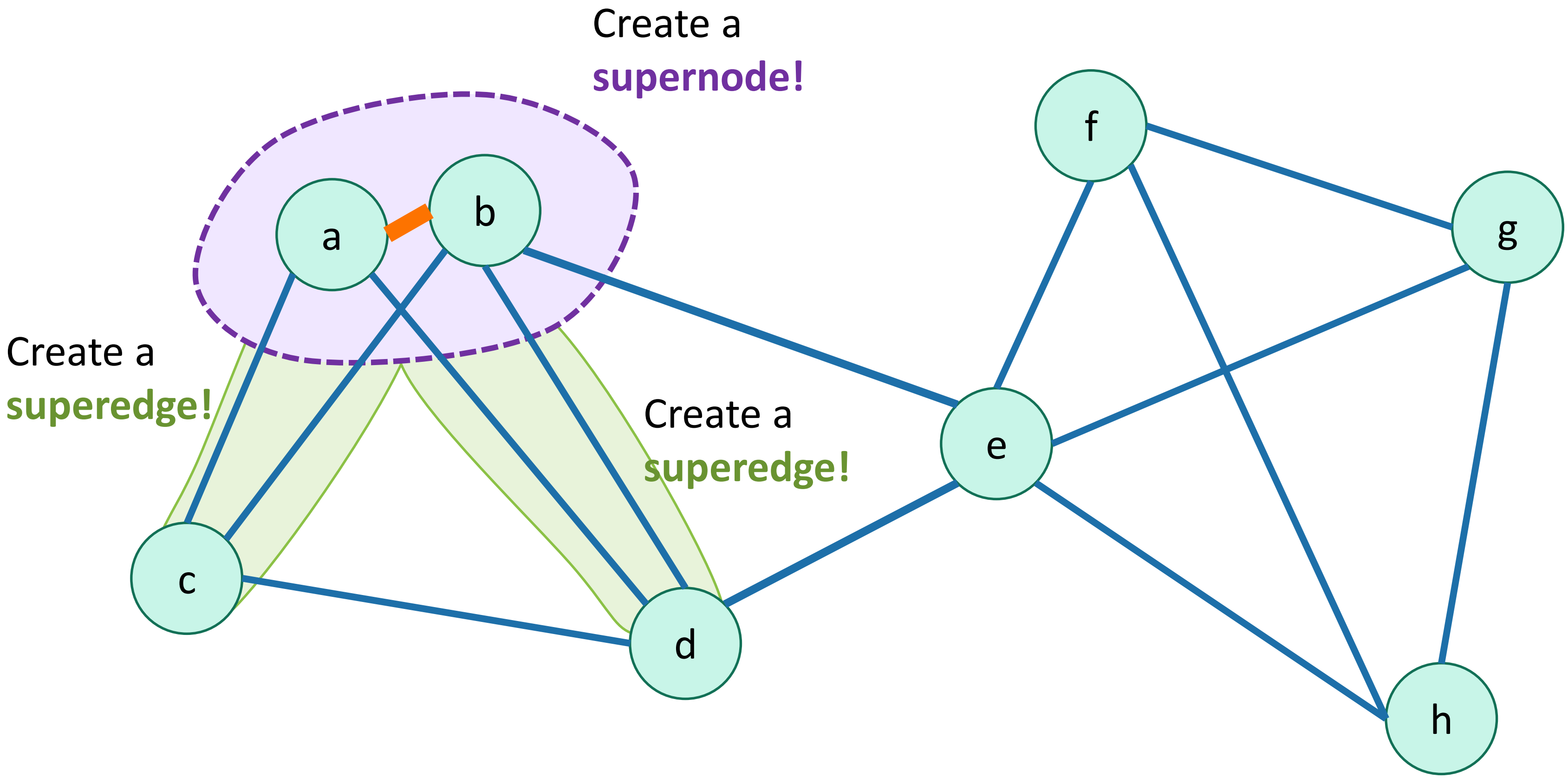
Probability that we didn't mess up:

12/14

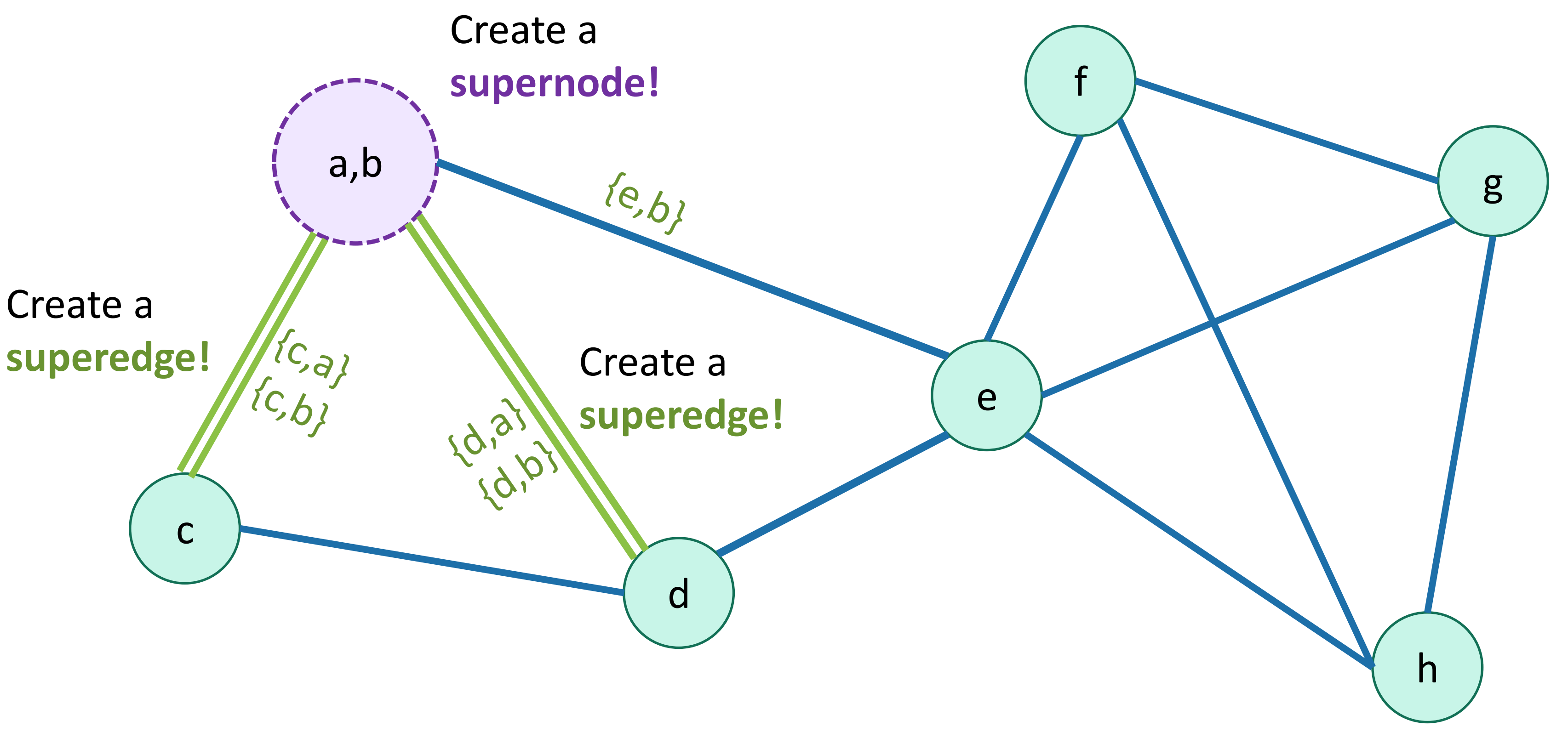
There are 14 edges, 12 of which are good to contract.



Karger’s algorithm



Karger’s algorithm

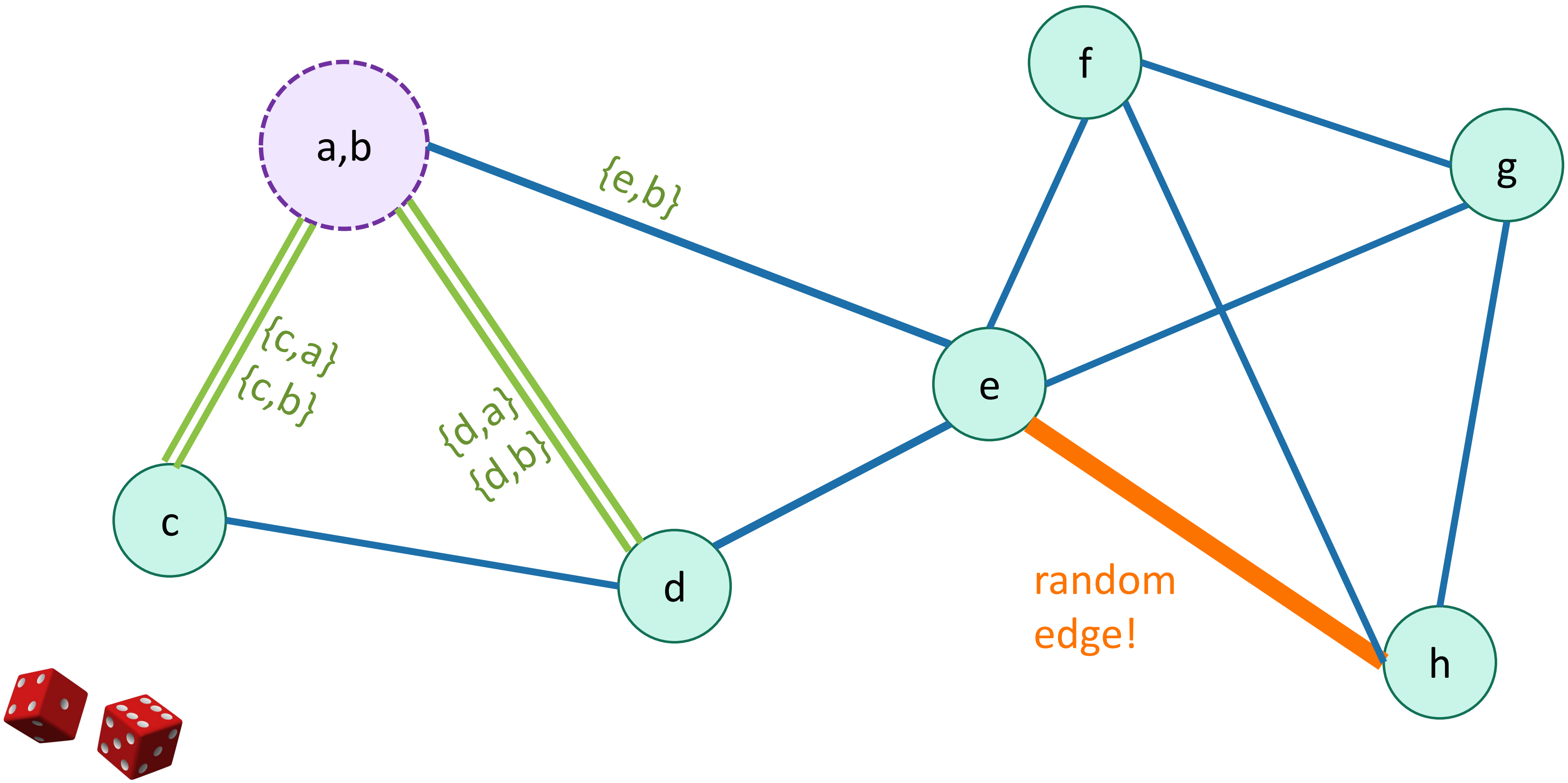


Karger's algorithm

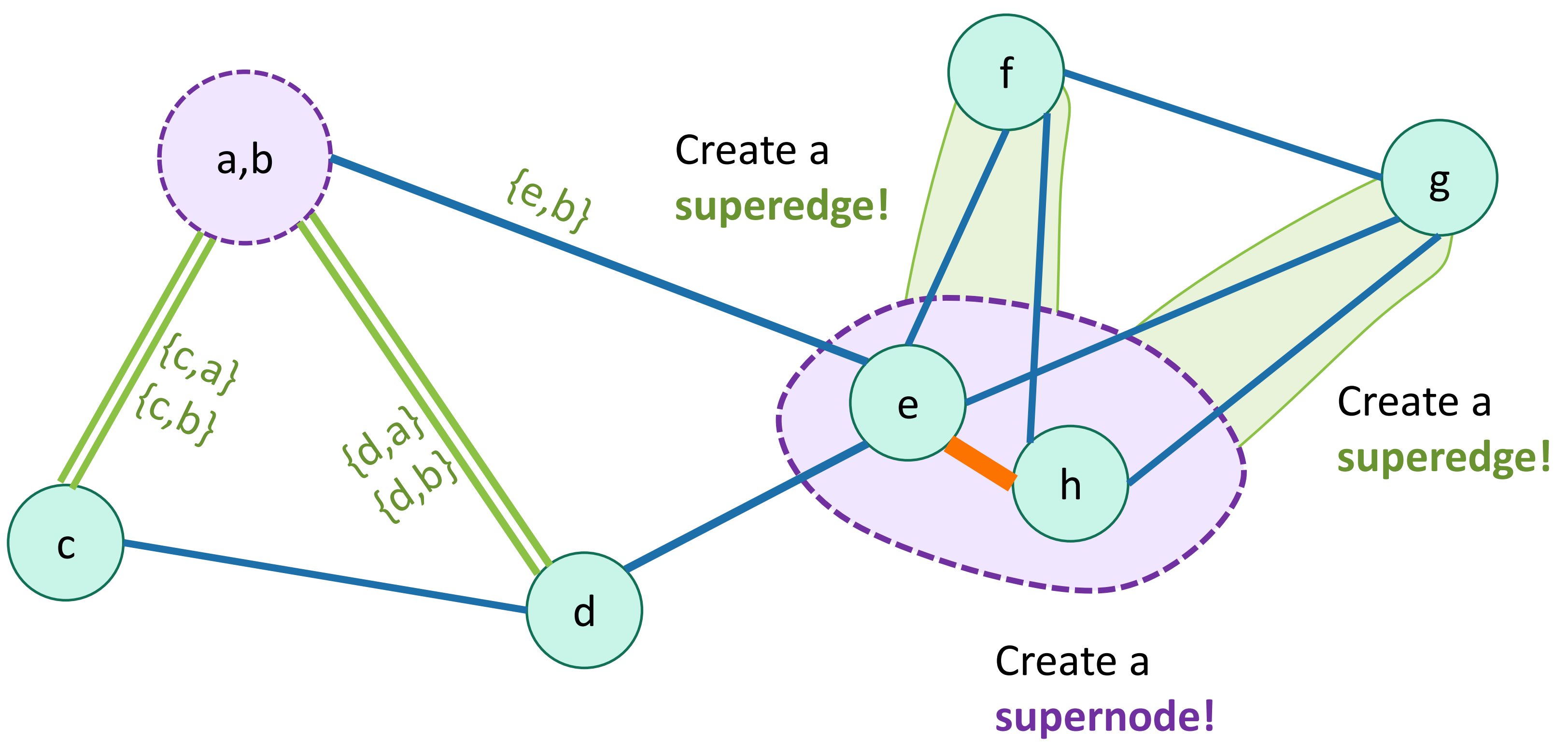
Probability that we didn't mess up:

11/13

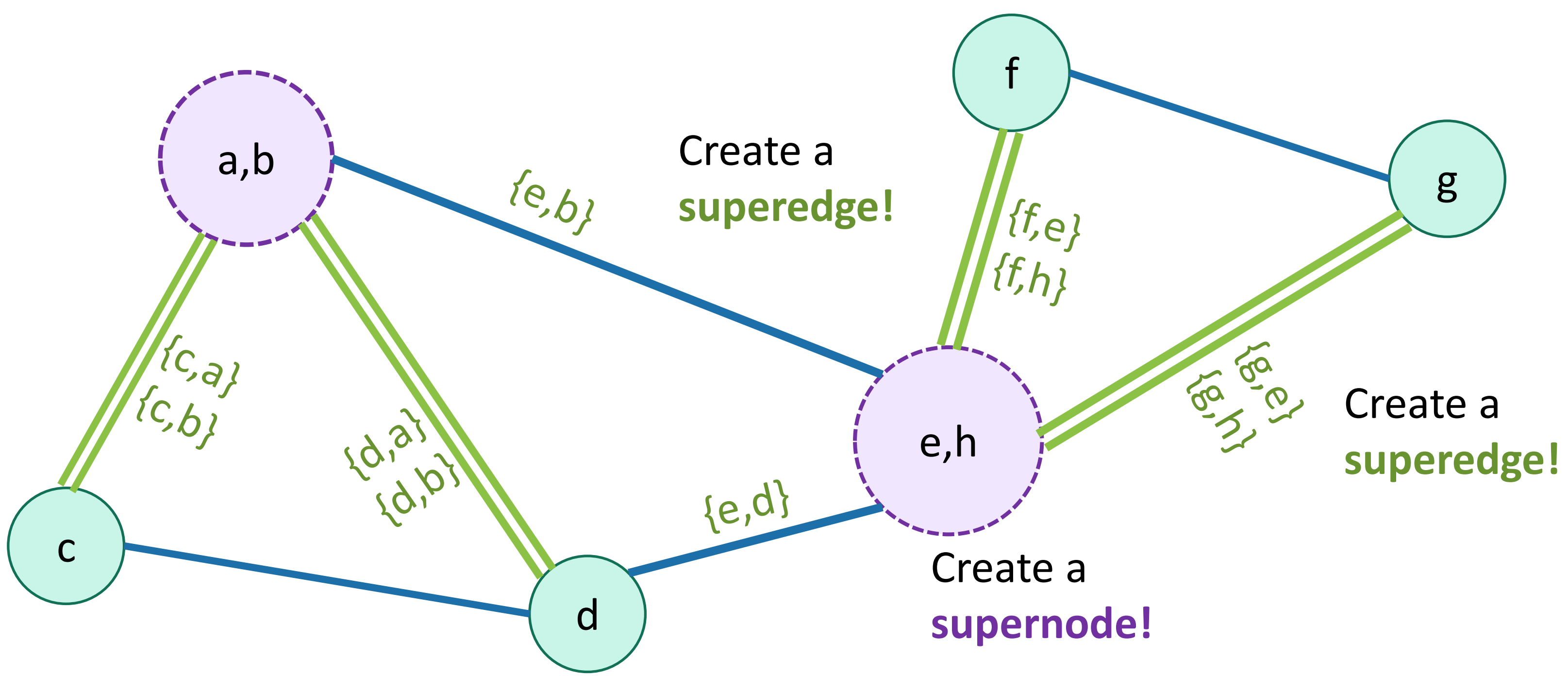
Now there are only 13 edges, since the edge between a and b disappeared.



Karger’s algorithm



Karger's algorithm

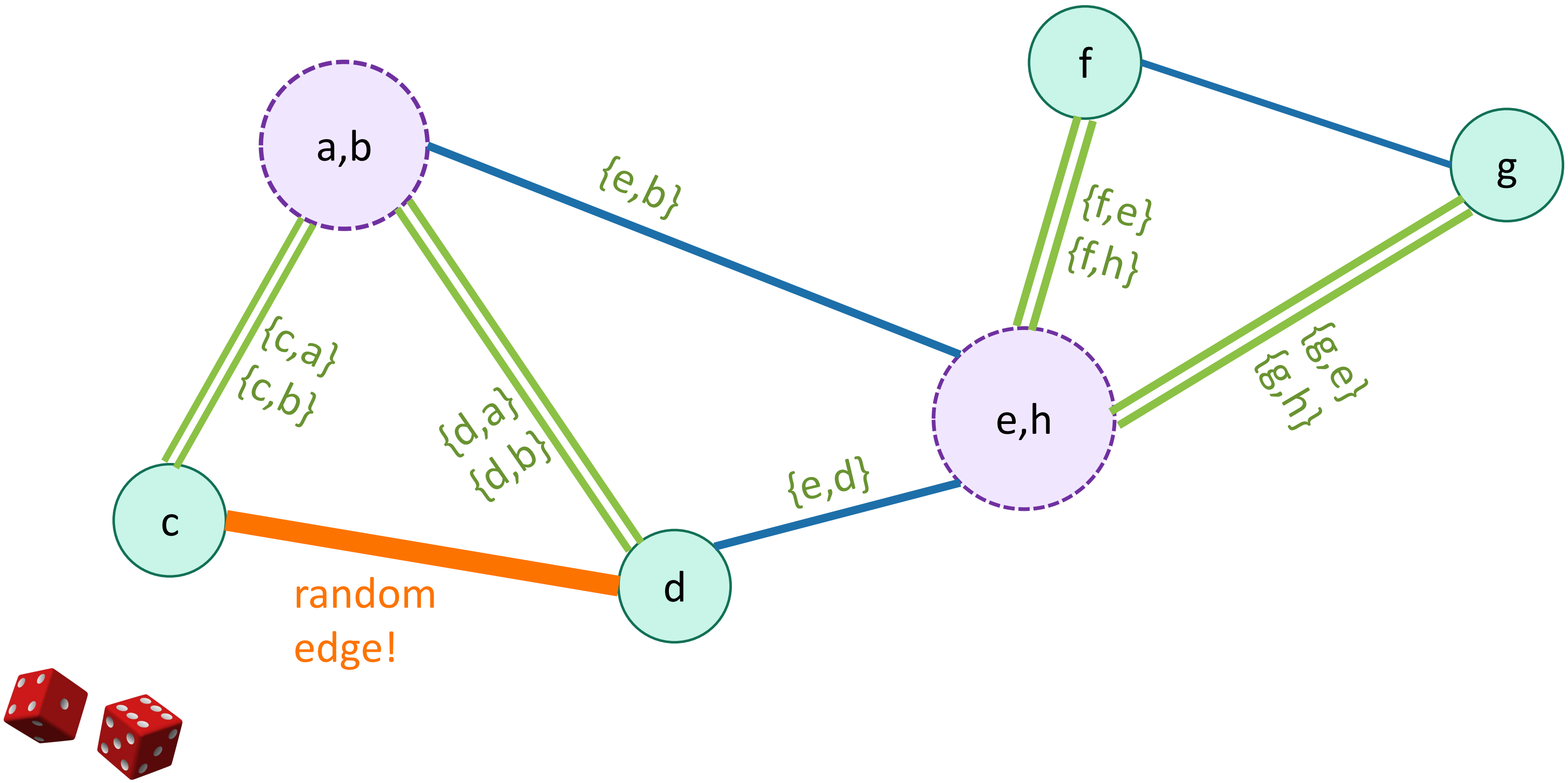


Karger's algorithm

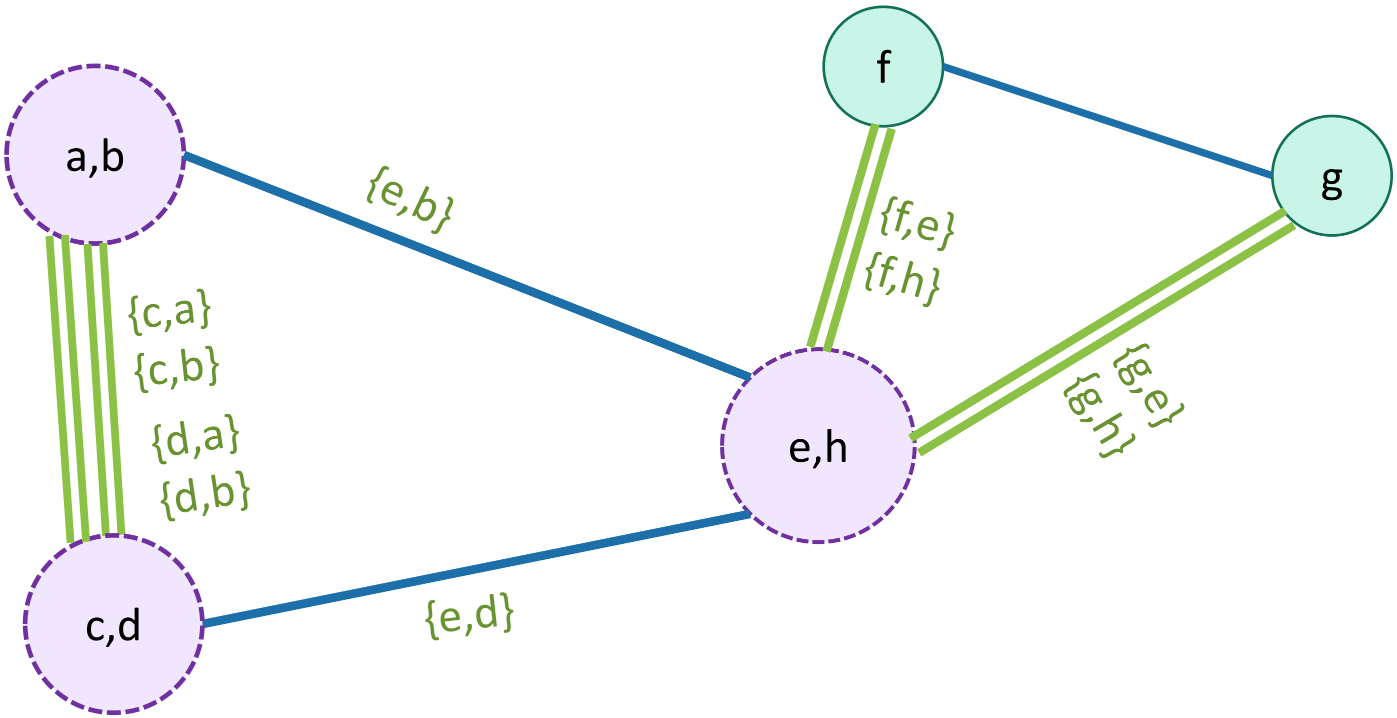
Probability that we didn't mess up:

10/12

Now there are only 12 edges, since the edge between e and h disappeared.



Karger’s algorithm

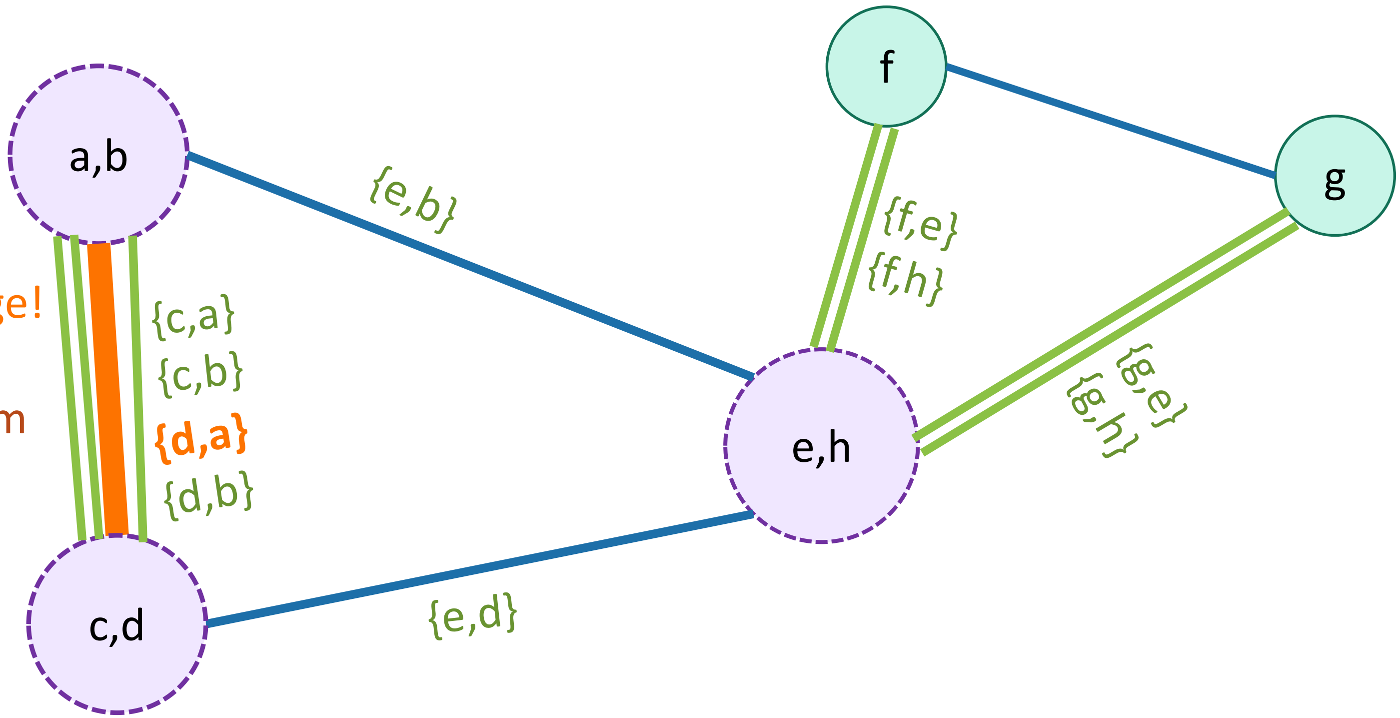


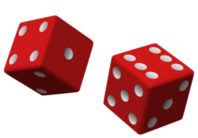
Karger’s algorithm

Probability that we didn’t mess up:

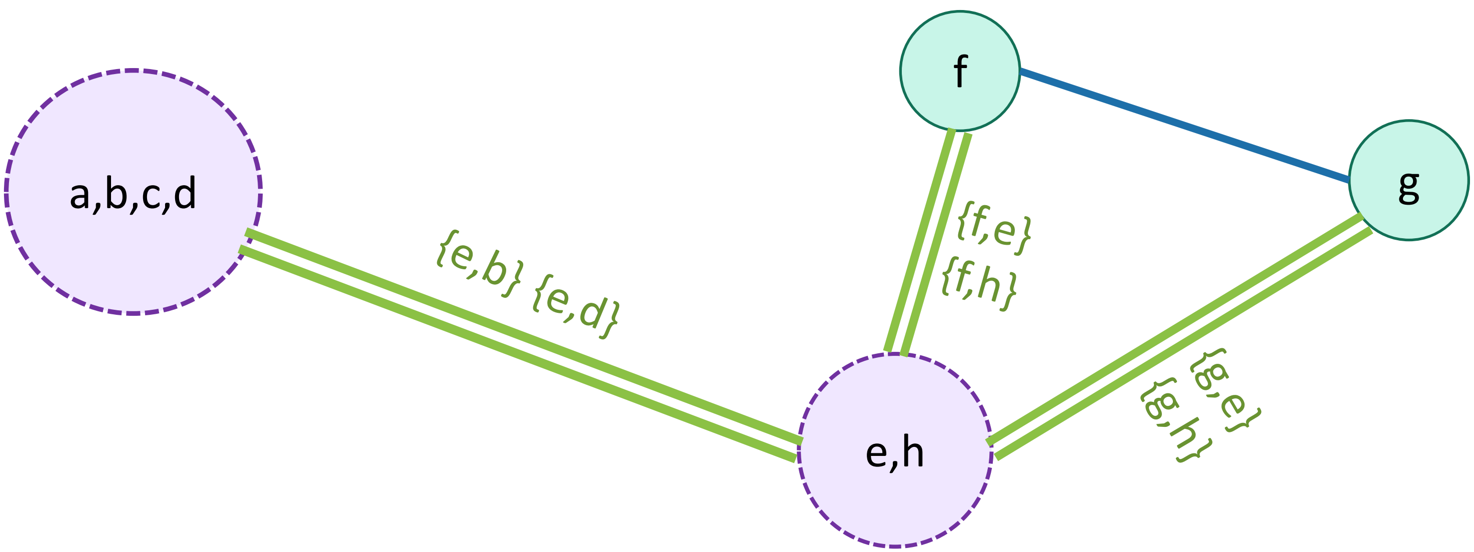
9/11

random edge!
(We pick at random from the original edges).





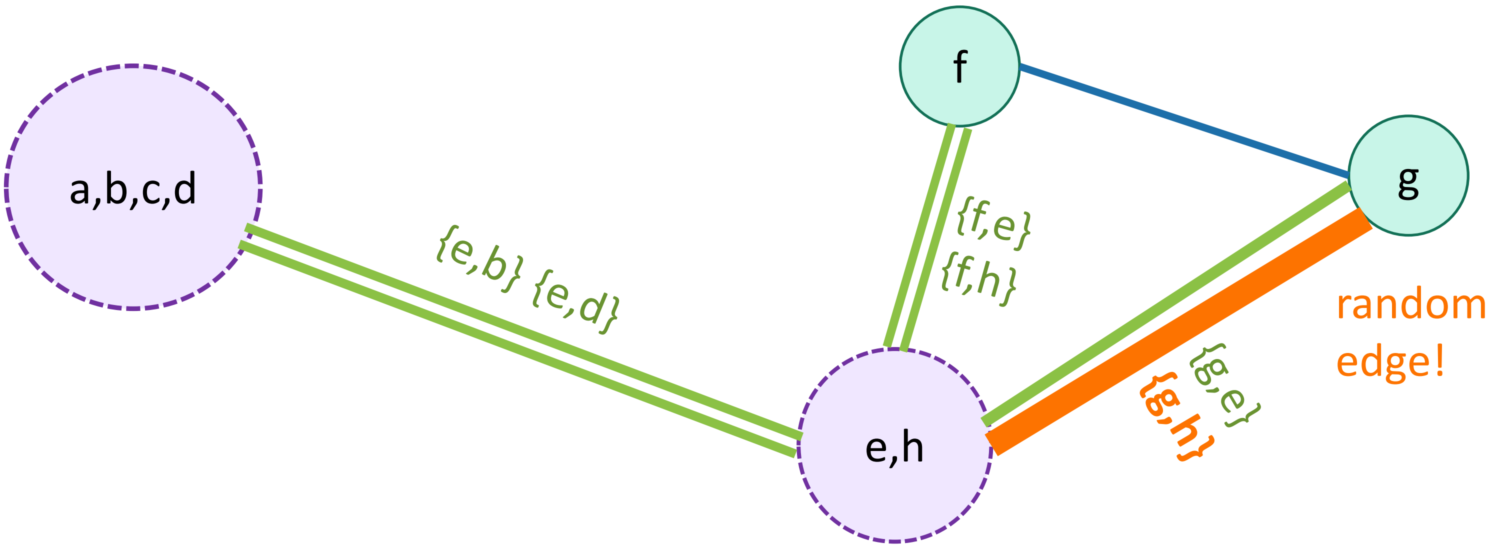
Karger's algorithm

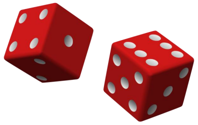


Karger's algorithm

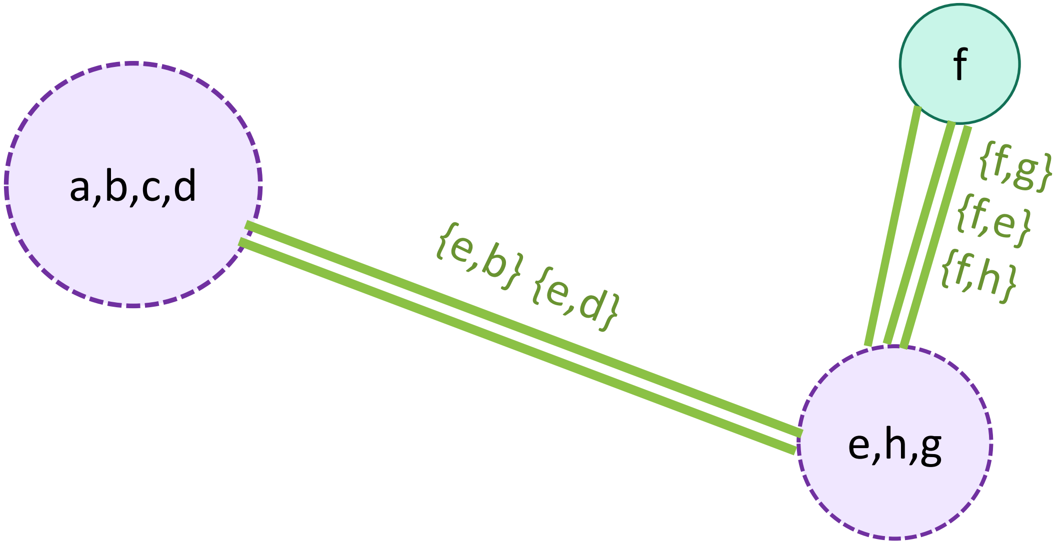
Probability that we didn't mess up:

5/7





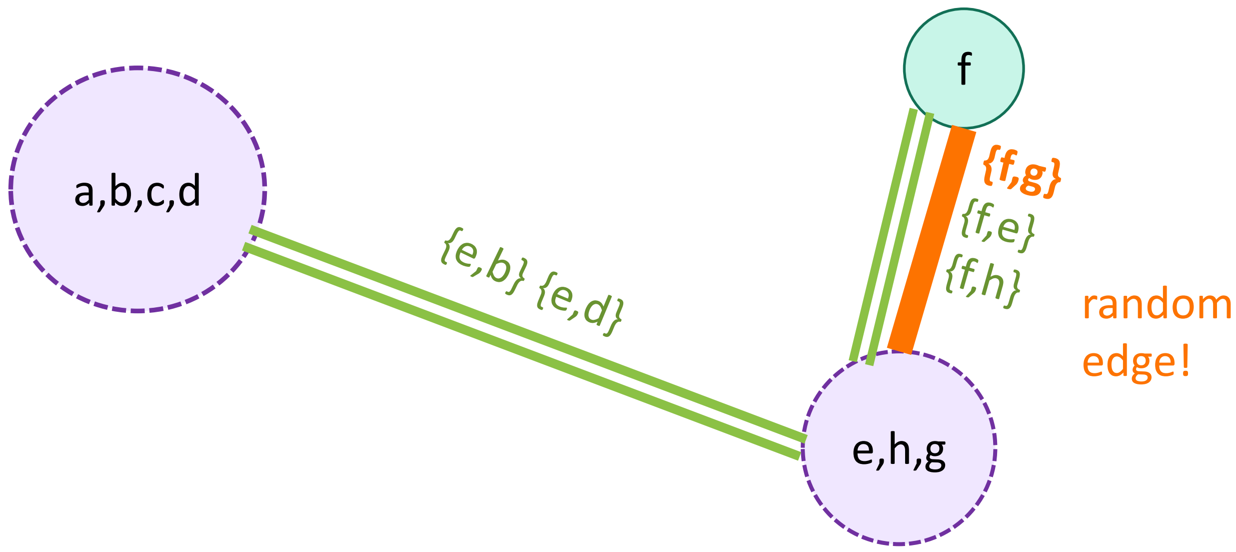
Karger’s algorithm



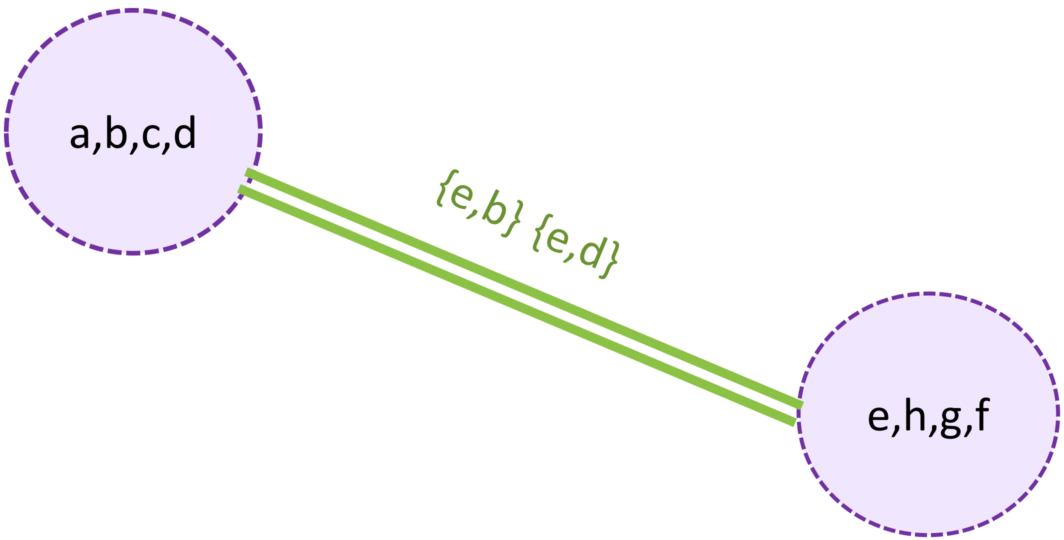
Karger's algorithm

Probability that we didn't mess up:

3/5



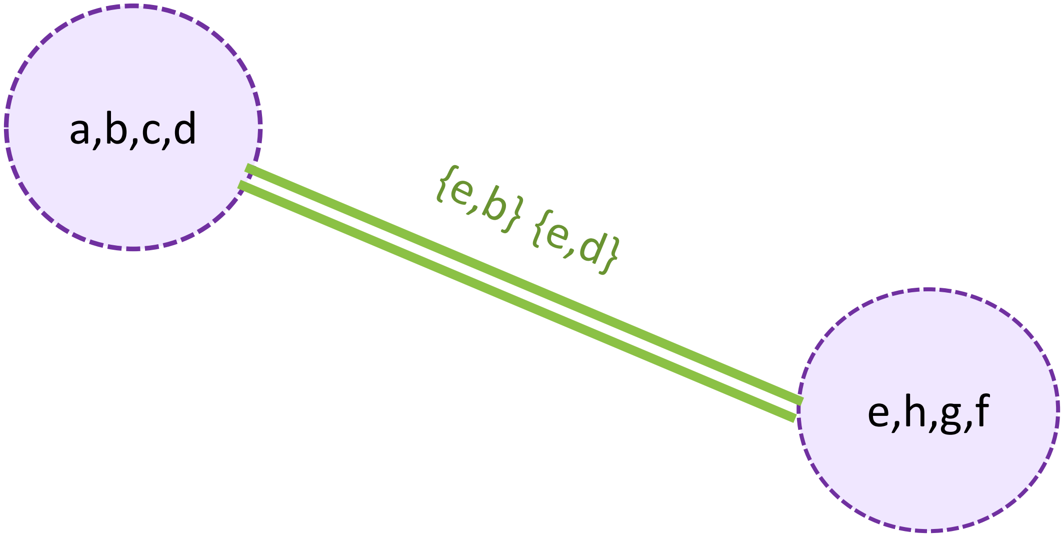
Karger’s algorithm



Karger's algorithm

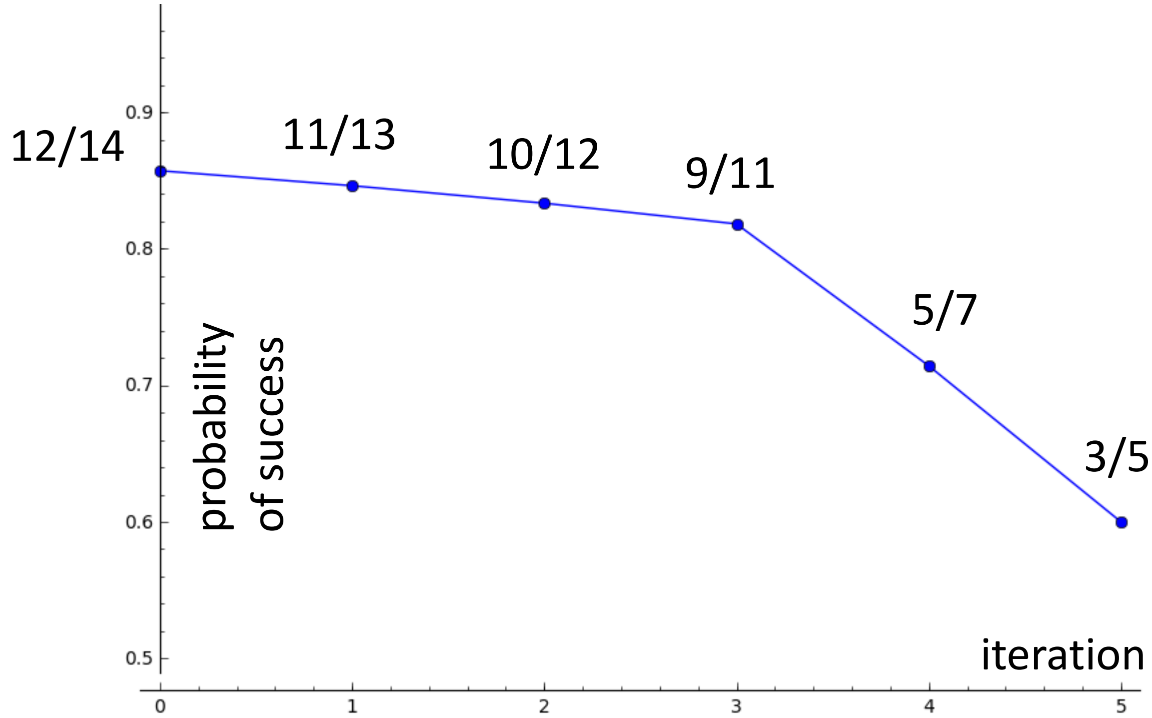
Now stop!

- There are only two nodes left.



Probability of not messing up

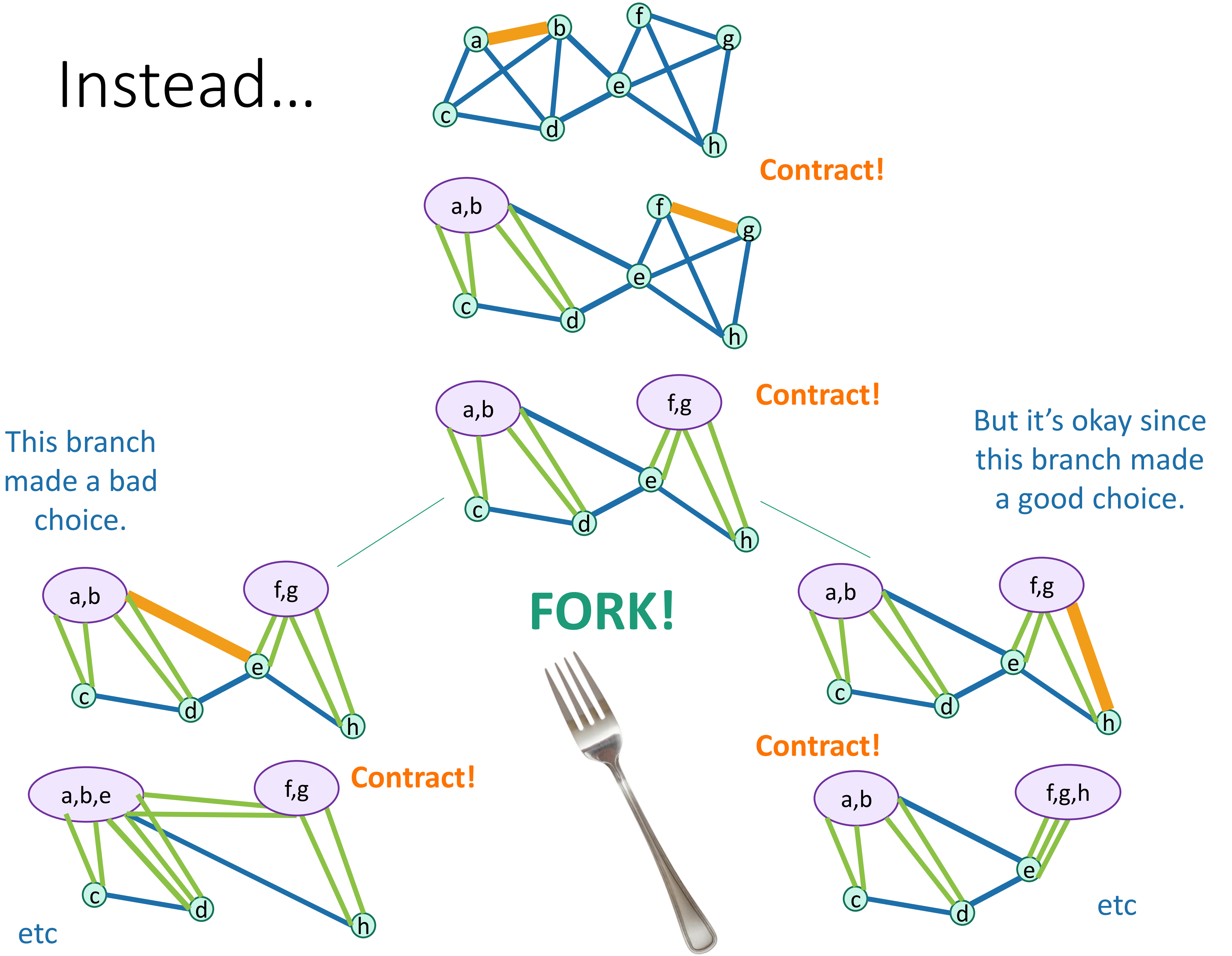
- At the beginning, it's pretty likely we'll be fine.
- The probability that we mess up gets worse and worse over time.



Moral:

Repeating the stuff from the beginning of the algorithm is **wasteful**!

Instead…



a
b
c
d
e
f
g
h
Contract!
a,b
f,g
This branch made a bad choice.
But it's okay since this branch made a good choice.
FORK!
a,b,e
f,g,h
etc
etc

In words

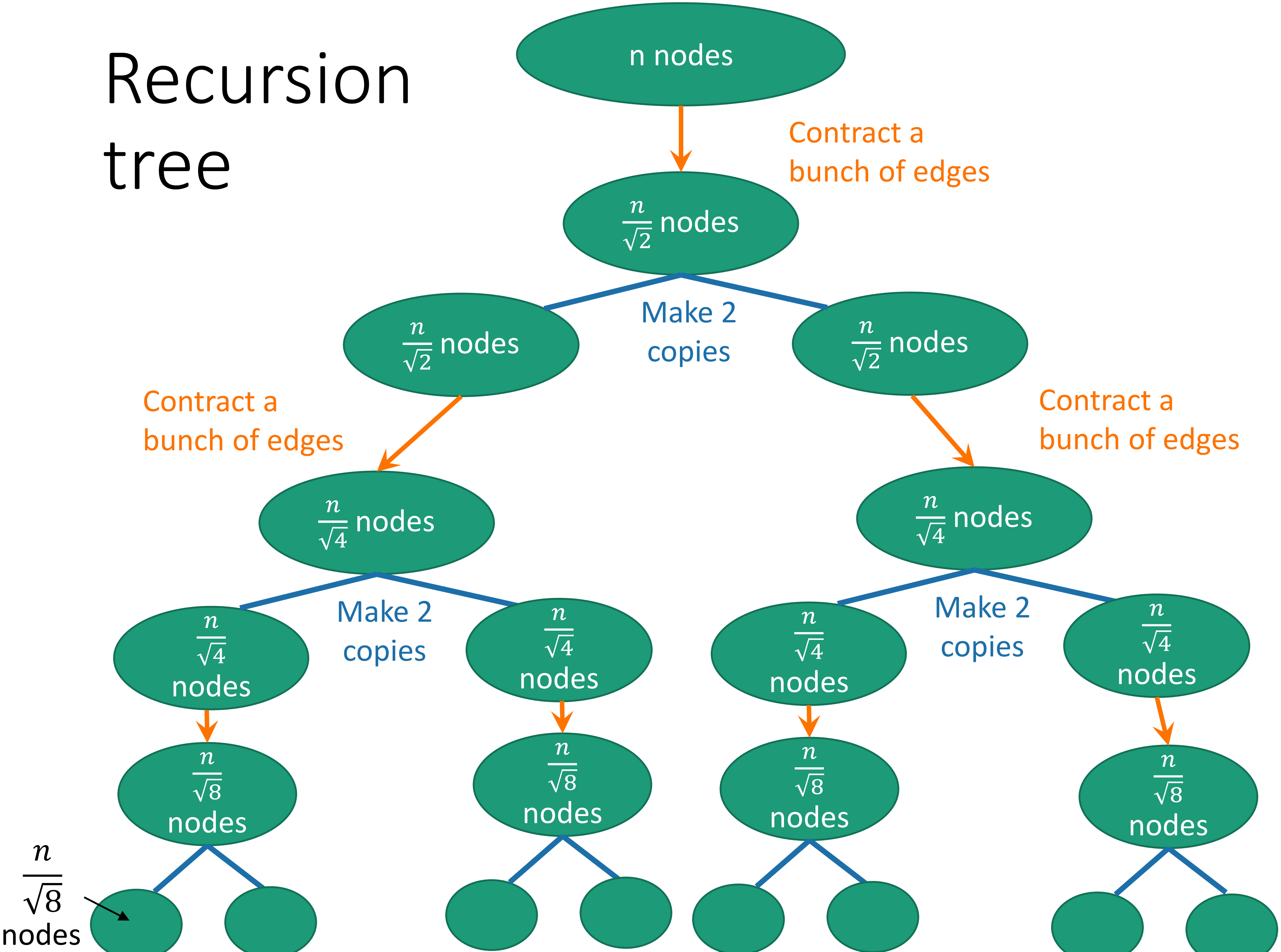
- Run Karger's algorithm on G for a bit.
    - Until there are $\frac{\mathrm{n}}{\sqrt{2}}$ supernodes left.
- Then split into two independent copies, G_1 and G_2
- Run Karger's algorithm on each of those for a bit.
    - Until there are $\frac{\left(\frac{\mathrm{n}}{\sqrt{2}}\right)}{\sqrt{2}} = \frac{n}{2}$ supernodes left in each.
- Then split each of those into two independent copies…

Why $\frac{\mathrm{n}}{\sqrt{2}}$? We'll see later.

In pseudocode

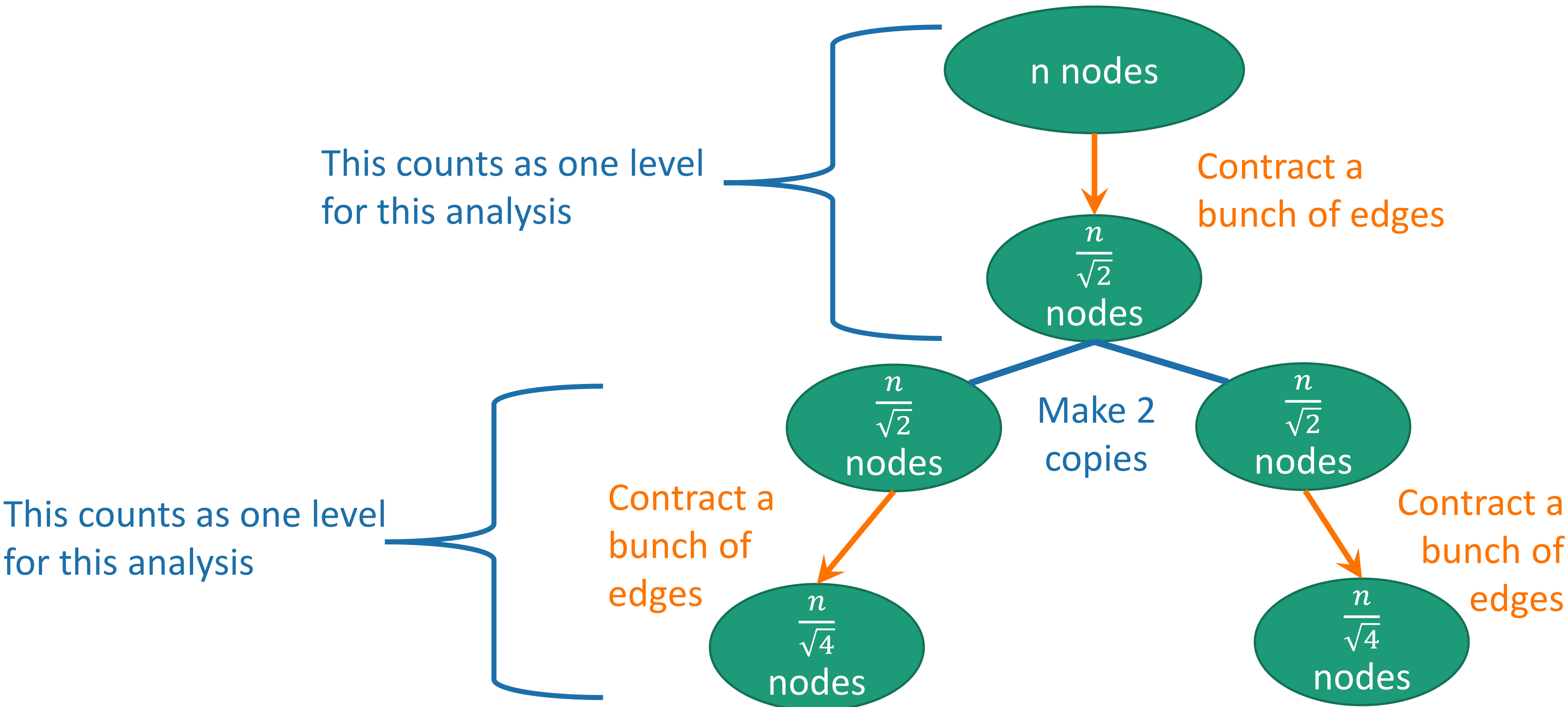
- **KargerStein**(G = (V,E)):
  - n ← |V|
  - if n < 4:
    - find a min-cut by brute force \\ time O(1)
  - Run Karger's algorithm on G with independent repetitions until $\left\lfloor \frac{n}{\sqrt{2}} \right\rfloor$ nodes remain.
  - G_1, G_2 ← copies of what's left of G
  - S_1 = **KargerStein**(G_1)
  - S_2 = **KargerStein**(G_2)
  - **return** whichever of S_1, S_2 is the smaller cut.

Recursion tree



Recursion tree

- depth is $\log_{\sqrt{2}}(n) = \frac{\log(n)}{\log(\sqrt{2})} = 2\log(n)$
- number of leaves is $2^{2\log(n)} = n^2$

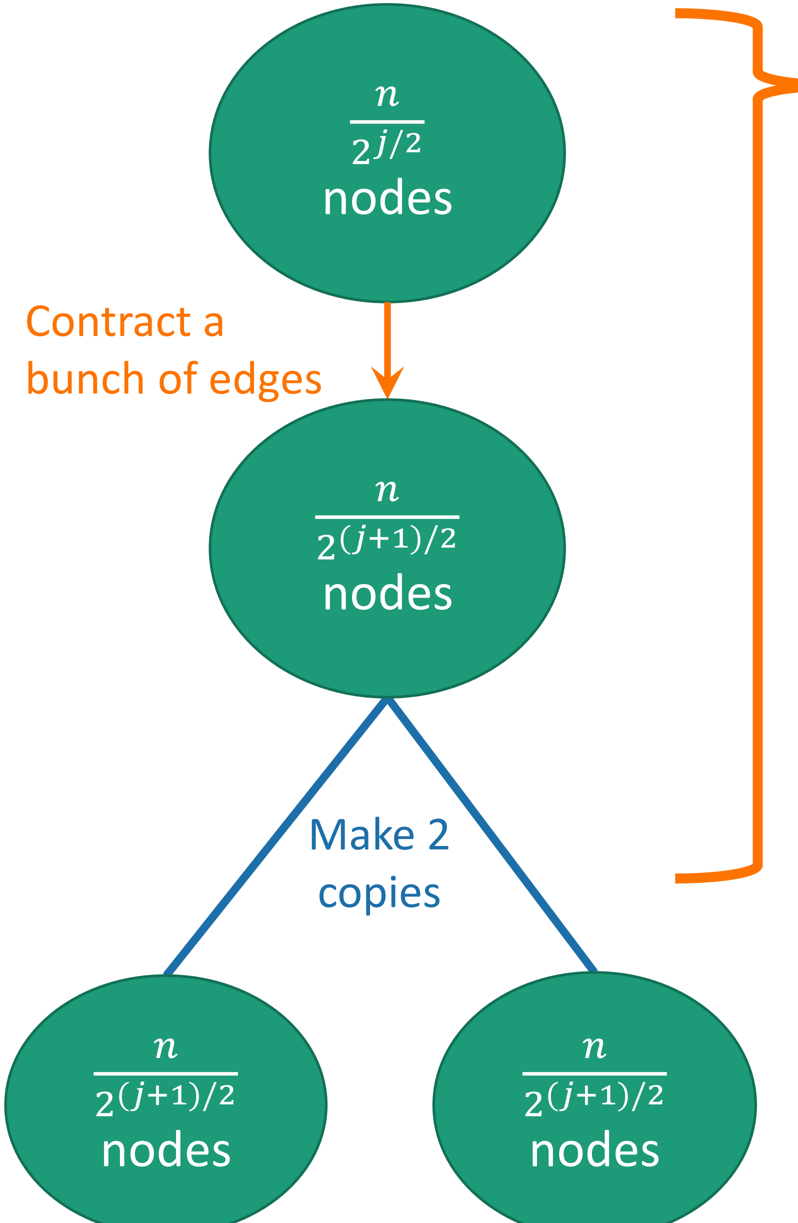


Two questions

- Does this work?

- Is it fast?

At the j^{th} level



- The amount of work per level is the amount of work needed to reduce the number of nodes by a factor of $\sqrt{2}$.
- That's at most $O(n^2)$.
  - since that's the time it takes to run Karger's algorithm once, cutting down the number of supernodes to **two**.
- Our recurrence relation is…

$$T(n) = 2T(n/\sqrt{2}) + O(n^2)$$

- The Master Theorem says…

$$T(n) = O(n^2 \log(n))$$

Jedi Master Yoda

Two questions

- Does this work?

- Is it fast?
  - Yes, $O(n^2 \log(n))$.

Why $n/\sqrt{2}$?

Suppose we contract n – t edges, until there are t supernodes remaining.

- Suppose the first n-t edges that we choose are

$$e_1, e_2, \ldots, e_{n-t}$$

- **PR**[none of the e_i cross S* (up to the n-t'th)]

  = **PR**[e_1 doesn't cross S*]

  $\times$ **PR**[e_2 doesn't cross S* | e_1 doesn't cross S*]

  …

  $\times$ **PR**[e_{n-t} doesn't cross S* | e_1,…,e_{n-t-1} don't cross S*]

Why $n/\sqrt{2}$?

Suppose we contract n – t edges, until there are t supernodes remaining.

- Suppose the first n-t edges that we choose are

$$e_1, e_2, \ldots, e_{n-t}$$

- **PR**[none of the e_i cross S* (up to the n-t'th)]

$$= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\left(\frac{n-5}{n-3}\right)\left(\frac{n-6}{n-4}\right)\cdots\left(\frac{t+1}{t+3}\right)\left(\frac{t}{t+2}\right)\left(\frac{t-1}{t+1}\right)$$

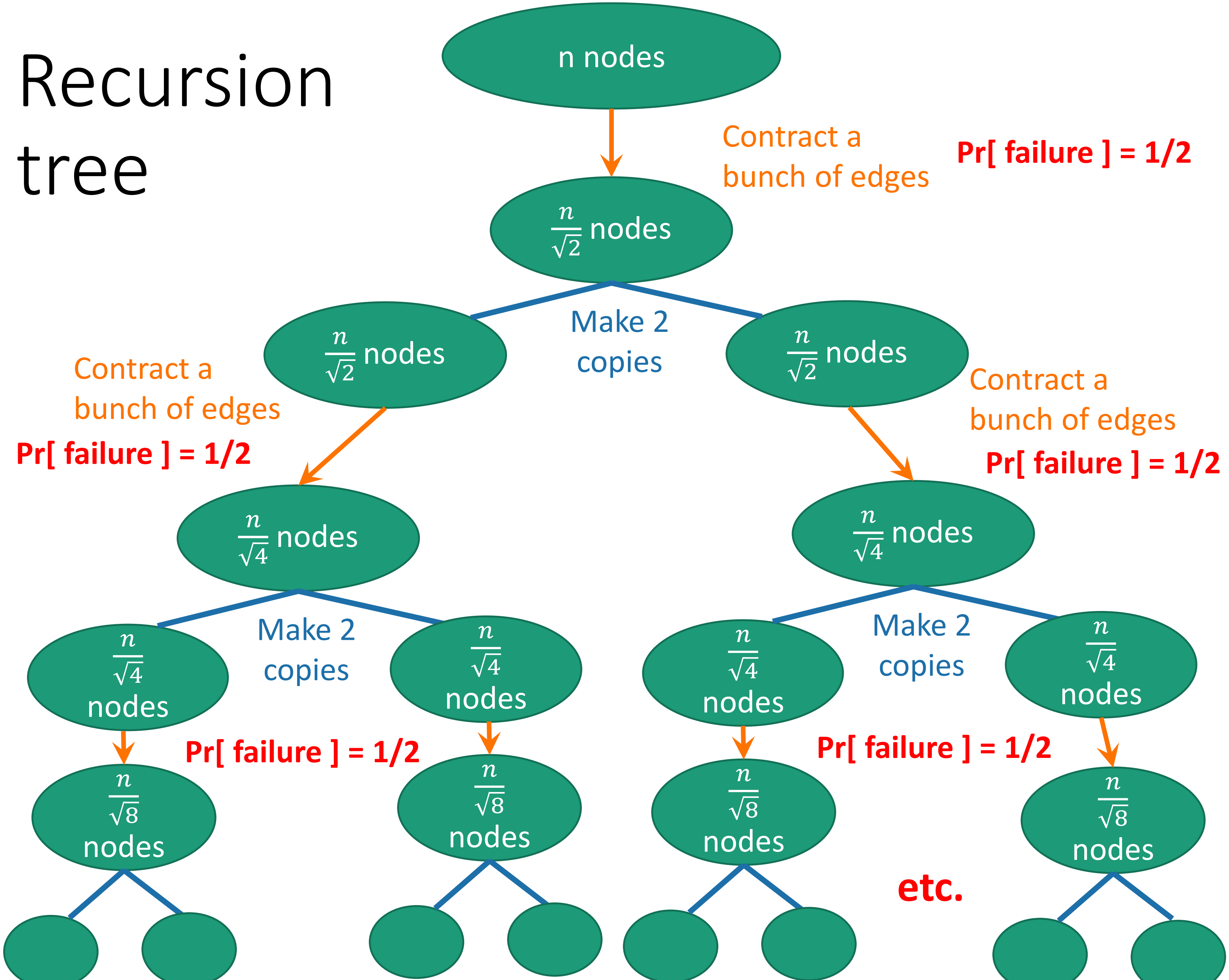
$$= \frac{t\cdot(t-1)}{n\cdot(n-1)}$$

Choose $\boldsymbol{t = n/\sqrt{2}}$

$$= \frac{\frac{n}{\sqrt{2}}\cdot\left(\frac{n}{\sqrt{2}}-1\right)}{n\cdot(n-1)} \approx \mathbf{\frac{1}{2}}$$

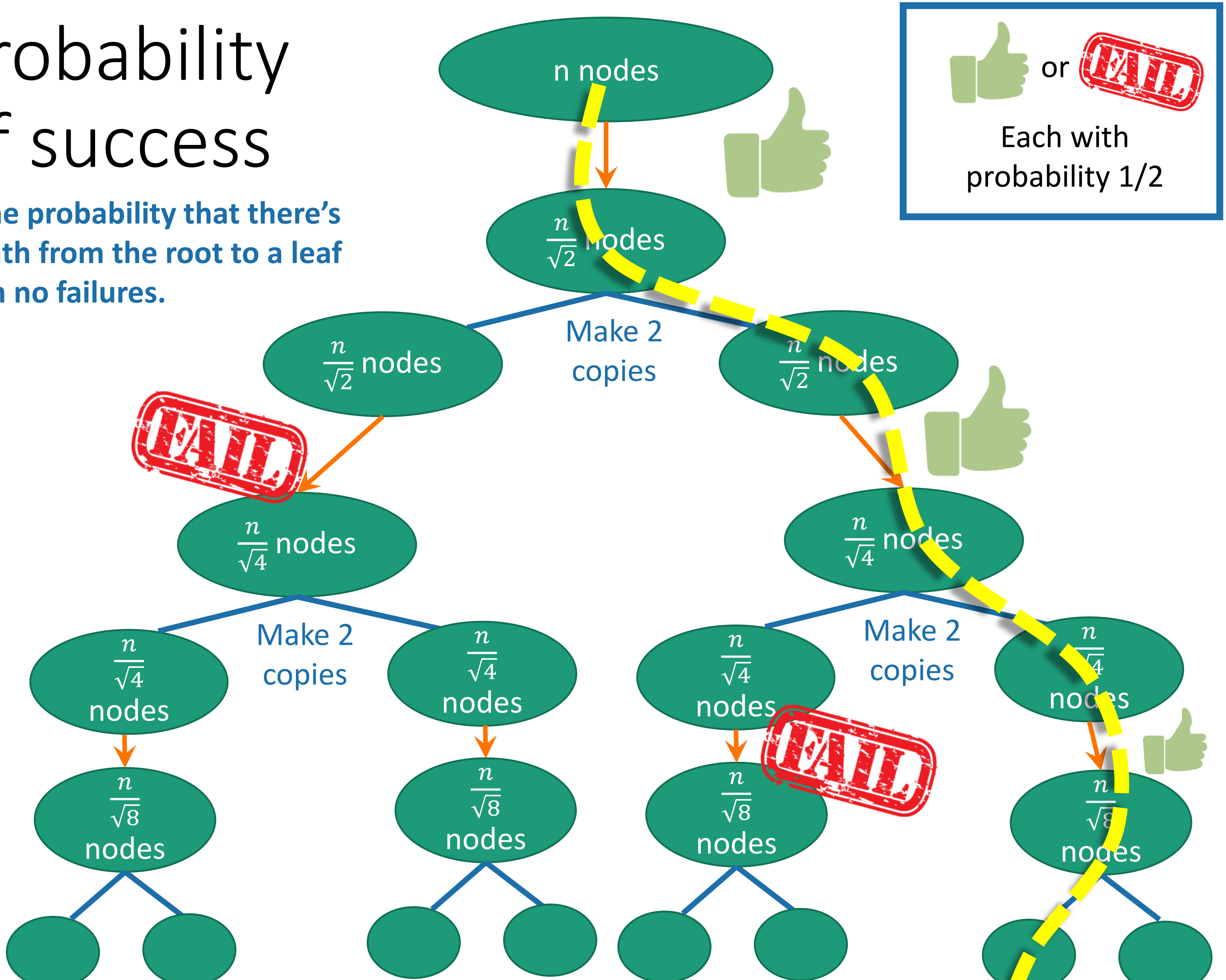
when n is large

Recursion tree



Probability of success

Is the probability that there's a path from the root to a leaf with no failures.

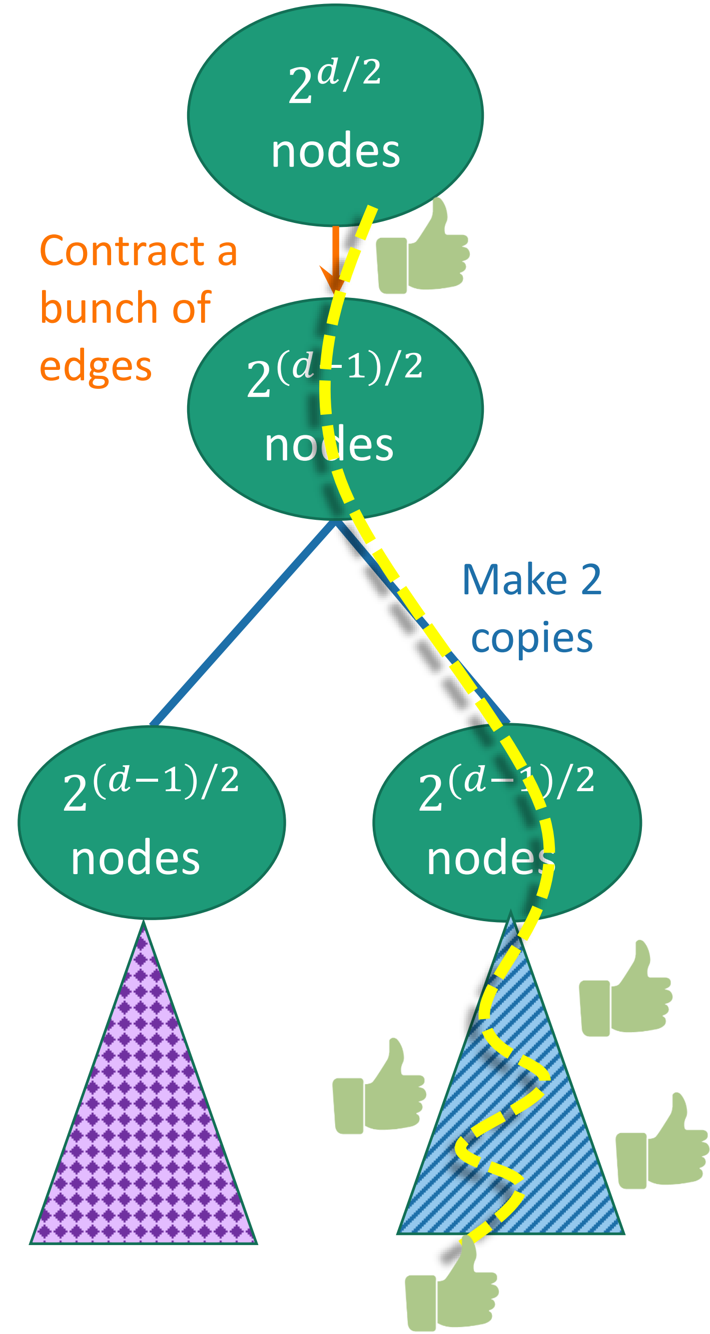


The problem we need to analyze

- Let T be binary tree of depth 2log(n)
- Each node of T succeeds or fails independently with probability 1/2
- What is the probability that there’s a path from the root to any leaf that’s entirely successful?

Analysis

- Say the tree has height d.
- Let $\boldsymbol{p_d}$ be the probability that there's a path from the root to a leaf that **doesn't fail**.
- $p_d = \frac{1}{2} \cdot \Pr\left[\text{at least one subtree has a successful path}\right]$
- $= \frac{1}{2} \cdot \left(\Pr\left[\text{wins}\right] + \Pr\left[\text{wins}\right] - \Pr\left[\text{both win}\right] \right)$
- $= \frac{1}{2} \cdot \left(p_{d-1} + p_{d-1} - p_{d-1}^2 \right)$
- $= p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2$



It’s a recurrence relation!

- $p_d = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2$
- $p_0 = 1$

- We are real good at those.
- In this case, the answer is:
  - **Claim**: for all d, $p_d \geq \frac{1}{d+1}$

Recurrence relation

- $p_d = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2$
- $p_0 = 1$

- **Claim**: for all d, $p_d \geq \frac{1}{d+1}$
- **Proof**: induction on d.
  - **Base case**: $1 \geq 1$. **YEP.**
  - **Inductive step**: say d > 0.
    - Suppose that $p_{d-1} \geq \frac{1}{d}$.
    - $p_d = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2$
    - $\geq \frac{1}{d} - \frac{1}{2} \cdot \frac{1}{d^2}$
    - $\geq \frac{1}{d} - \frac{1}{d(d+1)}$
    - $= \frac{1}{d+1}$

This slide skipped in class

What does that mean for Karger-Stein?

Claim: for all d, $p_d \geq \frac{1}{d+1}$

- For d = 2log(n)
  - that is, d = the height of the tree:

$$p_{2\log(n)} \geq \frac{1}{2\log(n) + 1}$$

- aka,

Pr[Karger-Stein is successful] = $\Omega\left(\frac{1}{\log(n)}\right)$

Altogether now

- We can do the same trick as before to amplify the success probability.
  - Run Karger-Stein $O\left(\log(n) \cdot \log\left(\frac{1}{\delta}\right)\right)$ times to achieve success probability $1 - \delta$.
- Each iteration takes time $O(n^2 \log(n))$
  - That's what we proved before.
- Choosing $\delta = 0.01$ as before, the total runtime is

$$O\left(n^2 \log(n) \cdot \log(n)\right) = O(n^2 \log(n)^2)$$

Much better than $O(n^4)$!

What have we learned?

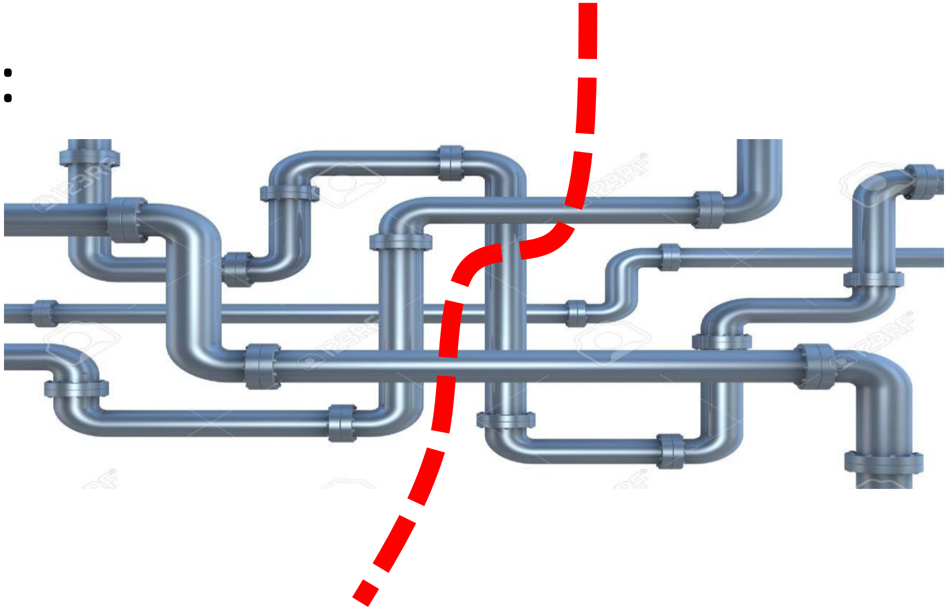
- Just repeating Karger's algorithm isn't the best use of repetition.
  - We're probably going to be correct near the beginning.
- Instead, Karger-Stein repeats when it counts.
  - If we wait until there are $\frac{n}{\sqrt{2}}$ nodes left, the probability that we fail is close to ½.
- This lets us find a global minimum cut in an undirected graph in time **$O(n^2 \log^2(n))$.**
  - Notice that we can't do better than n^2 in a dense graph (we need to look at all the edges), so this is pretty good.

Recap

- Some algorithms:
  - Karger's algorithm for global min-cut
  - Improvement: Karger-Stein
- Some concepts:
  - Monte Carlo algorithms:
    - Might be wrong, are always fast.
  - We can boost their success probability with repetition.
  - Sometimes we can do this repetition very cleverly.

Next time

- Another sort of min-cut:
  - s-t min-cut
  - also max-flow!



NEXT LECTURE

- Network Flow
- Max-Flow, Min-cut
- Ford-Fulkerson Algorithm

Week	Date	Topics
1	22 Feb	Introduction. Some representative problems
2	1 March	Stable Matching
3	8 March	Basics of algorithm analysis.
4	15 March	Graphs (Project 1 announced)
5	22 March	Greedy algorithms I
6	29 March	Greedy algorithms II (Project 2 announced)
7	5 April	Divide and conquer
8	12 April	Midterm
9	19 April	Dynamic Programming I
10	26 April	Dynamic Programming II (Project 3 announced)
11	3 May	BREAK
12	10 May	Network Flow-I
13	17 May	Network Flow II
14	24 May	NP and computational intractability I
15	31 May	NP and computational intractability II