Lecture 16
Min Cut and Karger’s Algorithm
Announcements

• HW 7 due Friday
• HW 8 released Friday
  • Psych! There is no HW8.

• FINAL EXAM:
  • Wednesday December 13
  • 3:30 – 6:30pm
CS 83 - PLAYBACK THEATER FOR RESEARCH

A FEEL GOOD COURSE
Last time

• Minimum Spanning Trees!
  • Prim’s Algorithm
  • Kruskal’s Algorithm
Today

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

• Back to randomized algorithms!
Recall: cuts in graphs

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

*For today, all graphs are undirected and unweighted.
Recall: cuts in graphs

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

*For today, all graphs are undirected and unweighted.
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 week we’ll talk about **min s-t cuts**

• 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
Karger’s algorithm

- Finds **global minimum cuts** in undirected graphs
- Randomized algorithm
  - Monte Carlo, not Las Vegas
- Karger’s algorithm **might be wrong**.
  - While QuickSort 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.

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 $\ldots$ over the randomness in Karger’s algorithm, $\text{Karger}(G)$ returns a minimum cut.

• For all inputs $G$, with probability 1 Karger’s algorithm runs in time no more than $\ldots$.

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

random edge!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!
Karger’s algorithm

random edge!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!

Create a supernode!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!
Karger’s algorithm

random edge!
Karger’s algorithm
Karger’s algorithm
Karger’s algorithm
Karger’s algorithm

{e,b} {e,d} {f,e} {f,h} {g,e} {g,h} random edge!
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 Lecture 16 IPython Notebook for one way
  • This maintains a secondary “superGraph” which keeps track of superNodes and superEdges
  • There’s a hidden slide with pseudocode

• 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 a bit better with fancy data structures but this is good enough for now.
Pseudocode

• **Karger( G=(V,E) ):**
  
  - $\Gamma = \{ \text{SuperNode}(v) : v \in V \}$  
    // 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 $E$.  
    // we’ll choose randomly from $F$
  - $F = \text{copy of } E$
  - while $|\Gamma| > 2$:
    - $(u,v) \leftarrow$ 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.

• **merge( u, v ):**  
  // merge also knows about $\Gamma$ and the $E_{\overline{u},\overline{v}}$’s
  - $\overline{x} = \text{SuperNode}(\overline{u} \cup \overline{v})$
    // 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 $\Gamma$ and add $\overline{x}$.

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

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

**merge** takes time $O(n)$ naively

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?
  • About $O(n^2)$ if we don’t want to be too clever about it.
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
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:

The algorithm is only correct about 20% of the time!
But this is better than it could be

- Suppose that we chose cuts \textit{uniformly at random}.
  - That is, pick a random way to split the vertices into 2 parts.
But this is better than it could be

- 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
Much better than random!

Karger is correct about 20% of the time

Completely random is correct about 0.6% of the time
What’s going on?

• 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.

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

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

B: Actually, this cut can’t! (Because how would a and g end up in the same super-node?)

Thing 2: By only contracting edges we are ignoring certain cuts.
Why does that help?

• 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 bunch of times I’ll eventually get it right.
Thought experiment from pre-lecture exercise

• 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 \(a, b, c, d, e\)?

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?

[On board]
This is approximately what’s on the board

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

\[ E[ \text{we push the button until we get the minimum value} ] = \frac{1}{0.20} = 5 \]

This one we’ve done less frequently:

\[ \text{We push the button } t \text{ times and don’t ever get the min} \]
\[ \Pr[ = (1 - 0.2)^t \approx 0.32 \]

\[ \text{We push the button } 5 \text{ times and don’t ever get the min} \]
\[ \Pr[ ] = (1 - 0.2)^5 \approx 0.32 \]

\[ \text{We push the button } 10 \text{ times and don’t ever get the min} \]
\[ \Pr[ ] = (1 - 0.2)^{10} \approx 0.1 \]
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!
- 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!
If we’re going to repeat a bunch of times Karger is “better” than random.

- **Karger**: repeat on the order of $1/0.2 = 5$ times
- **Completely random**: repeat on the order of $1/0.008 = 800$ 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 “on the order of 5 times” statement.
Questions

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

\[ \text{at least } \frac{1}{\binom{n}{2}} \]

In this case, \( \frac{1}{\binom{8}{2}} \approx 0.036 \), so we are guaranteed to win at least 3.6% of the time.
Before we prove the Claim

2. How many times should we run Karger’s algorithm to succeed with probability 0.99?
A computation

Suppose:

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

$\Pr[\text{don't return a min cut in } T \text{ trials } ] = (1 - p)^T$

So $p = 1/\binom{n}{2}$ by the Claim. Let's choose $T = \binom{n}{2} \ln(1/\delta)$.

$\Pr[\text{don't return a min cut in } T \text{ trials } ]$

- $= (1 - p)^T$
- $\leq (e^{-p})^T$
- $= e^{-pT}$
- $= e^{-\ln(\frac{1}{\delta})}$
- $= \delta$

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

$1 - p \leq e^{-p}$
Theorem
Assuming the claim about $1/(n) ...$

• Suppose $G$ has $n$ vertices.

• Consider the following algorithm:
  • $\text{bestCut} = \text{None}$
  • $\textbf{for } t = 1, ..., \frac{n}{2}\ln \left(\frac{1}{\delta}\right)$ :
    • $\text{candidateCut} \leftarrow \text{Karger}(G)$
    • $\textbf{if } \text{candidateCut}$ is smaller than $\text{bestCut}$:
      • $\text{bestCut} \leftarrow \text{candidateCut}$
  • $\textbf{return } \text{bestCut}$

• Then $\Pr[\text{this doesn’t return a min cut}] \leq \delta$. 
What’s the running time?

- **Depends** on how we implement Karger’s algorithm.
- As stated, $O \left( n^2 \cdot \binom{n}{2} \ln \left( \frac{1}{\delta} \right) \right) = O(n^4)$

- If we use **union-find data structures**, we can do better.

These are the things we used to implement Kruskal’s algorithm last week.

Let’s go with $O(n^4)$ for now.

Write pseudocode for a fast version of Karger’s algorithm! How fast can you make the asymptotic running time?

Ollie the over-achieving ostrich
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 \( \frac{1}{\binom{n}{2}} \).
Now let’s prove that claim
Say that S* is a minimum cut.

- Suppose the edges that we choose are $e_1, e_2, ..., e_{n-2}$
- \( \text{PR[ return S* ]} = \text{PR[ none of the } e_i \text{ cross S* ]} \)
  \( = \text{PR[ } e_1 \text{ doesn’t cross S* ]} \)
  \( \times \text{PR[ } e_2 \text{ doesn’t cross S* | } e_1 \text{ doesn’t cross S* ]} \)
  ...
  \( \times \text{PR[ } e_{n-2} \text{ doesn’t cross S* | } e_1, ..., e_{n-3} \text{ don’t cross S* ]} \)
Focus in on:
\( \text{PR}[ e_j \text{ doesn’t cross } S^* \mid e_1,\ldots,e_{j-1} \text{ don’t cross } S^* ] \)

- Suppose: After \( j-1 \) iterations, we haven’t messed up yet!
- What’s the probability of messing up now?
Focus in on:

\[ \text{PR}[ e_j \text{ doesn’t cross } S^* \mid e_1, \ldots, e_{j-1} \text{ don’t cross } S^* ] \]

- 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 \( k \).

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

\[
\frac{k}{\binom{n-j+1}{2}k} = \frac{2}{n-j+1}
\]
Now let’s prove that claim
Say that $S^*$ is a minimum cut.

- Suppose the edges that we choose are $e_1, e_2, ..., e_{n-2}$
- $\text{PR} [ \text{return } S^* ] = \text{PR} [ \text{none of the } e_i \text{ cross } S^* ]$
  = $\text{PR} [ e_1 \text{ doesn’t cross } S^* ]$
  $\times \text{PR} [ e_2 \text{ doesn’t cross } S^* \mid e_1 \text{ doesn’t cross } S^* ]$
  ...
  $\times \text{PR} [ e_{n-2} \text{ doesn’t cross } S^* \mid e_1, ..., e_{n-3} \text{ don’t cross } S^* ]$
Now let’s prove that claim
Say that $S^*$ is a minimum cut.

- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- $\textsc{PR}[\text{return } S^* ] = \textsc{PR}[\text{none of the } e_i \text{ cross } S^* ]$
  $$= \left( \frac{n-2}{n} \right) \left( \frac{n-3}{n-1} \right) \left( \frac{n-4}{n-2} \right) \left( \frac{n-5}{n-3} \right) \left( \frac{n-6}{n-4} \right) \ldots \left( \frac{4}{6} \right) \left( \frac{3}{5} \right) \left( \frac{2}{4} \right) \left( \frac{1}{3} \right)$$
Now let’s prove that claim
Say that $S^*$ is a minimum cut.

- Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$
- $\text{PR[ return } S^* \text{ ]} = \text{PR[ none of the } e_i \text{ cross } S^* \text{ ]}$
  
  \[
  = \left( \frac{n-2}{n} \right) \left( \frac{n-3}{n-1} \right) \left( \frac{n-4}{n-2} \right) \left( \frac{n-5}{n-3} \right) \left( \frac{n-6}{n-4} \right) \ldots \left( \frac{4}{6} \right) \left( \frac{3}{5} \right) \left( \frac{2}{4} \right) \left( \frac{1}{2} \right)
  \]
  
  \[
  = \left( \frac{2}{n(n-1)} \right)
  \]
  
  \[
  = \frac{1}{\binom{n}{2}}
  \]

 CLAIM PROVED
Theorem

Assuming the claim about $1/(\binom{n}{2})$ ...

• Suppose G has n vertices.

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

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

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.

• 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 $O(n^4 \log(n))$. 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, but the important part is the high-level idea.

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: \( \frac{12}{14} \)

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

random edge!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!

Create a superedge!

Create a superedge!
Karger’s algorithm

Probability that we didn’t mess up: \( \frac{11}{13} \)

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

Create a supernode!

Create a superedge!

Create a superedge!
Karger’s algorithm

Create a supernode! Create a superedge! Create a superedge!
Karger’s algorithm

Probability that we didn’t mess up: \( \frac{10}{12} \)

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: \( \frac{5}{7} \)
Karger’s algorithm
Karger’s algorithm

Probability that we didn’t mess up:

3/5
Karger’s algorithm

\[ \{e, b\} \{e, d\} \]
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...

This branch made a bad choice.

But it’s okay since this branch made a good choice.

FORK!

Contract!

Contract!

Contract!

Contract!

etc
In words

• Run Karger’s algorithm on G for a bit.
  • Until there are $\frac{n}{\sqrt{2}}$ 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 $\frac{\left(\frac{n}{\sqrt{2}}\right)}{\sqrt{2}} = \frac{n}{2}$ supernodes left in each.

• Then split each of those into two independent copies...
In pseudocode

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

- n nodes
  - \( \frac{n}{\sqrt{2}} \) nodes
    - Make 2 copies
      - \( \frac{n}{\sqrt{4}} \) nodes
        - Make 2 copies
          - \( \frac{n}{\sqrt{8}} \) nodes
            - Contract a bunch of edges
          - \( \frac{n}{\sqrt{8}} \) nodes
            - Contract a bunch of edges
          - \( \frac{n}{\sqrt{8}} \) nodes
            - Contract a bunch of edges
          - \( \frac{n}{\sqrt{8}} \) nodes
            - Contract a bunch of edges

- \( n \) nodes
  - \( \frac{n}{\sqrt{2}} \) nodes
    - Contract a bunch of edges
  - \( \frac{n}{\sqrt{2}} \) nodes
    - Contract a bunch of edges
  - \( \frac{n}{\sqrt{2}} \) nodes
    - Contract a bunch of edges
  - \( \frac{n}{\sqrt{2}} \) nodes
    - Contract a bunch of edges
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, $O(n^2 \log(n))$. 
First

Why $n/\sqrt{2}$ ?

• Suppose the first $n-t$ edges that we choose are $e_1, e_2, \ldots, e_{n-t}$

• $\Pr[\text{none of the } e_i \text{ cross } S^* \text{ (up to the } n-t'\text{th}) ]$

  $= \Pr[\ e_1 \text{ doesn’t cross } S^* ]$

  $\times \Pr[\ e_2 \text{ doesn’t cross } S^* \mid e_1 \text{ doesn’t cross } S^* ]$

  $\cdots$

  $\times \Pr[\ e_{n-t} \text{ doesn’t cross } S^* \mid e_1, \ldots, e_{n-t-1} \text{ don’t cross } S^* ]$

Suppose we contract $n - t$ edges, until there are $t$ supernodes remaining.
First

Why \( n/\sqrt{2} \)?

• Suppose the first \( n-t \) edges that we choose are \( e_1, e_2, \ldots, e_{n-t} \)

• \textbf{PR} [ none of the \( e_i \) cross \( S^* \) (up to the \( n-t \)'th) ]

\[
\frac{(n-2)}{n} \cdot \frac{(n-3)}{n-1} \cdot \frac{(n-4)}{n-2} \cdot \frac{(n-5)}{n-3} \cdot \frac{(n-6)}{n-4} \cdots \frac{(t+1)}{t+3} \cdot \frac{t}{t+2} \cdot \frac{t-1}{t+1}
\]

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

Choose \( t = n/\sqrt{2} \)

\[
= \frac{\sqrt{2} \cdot (\sqrt{2}-1)}{n \cdot (n-1)} \approx \frac{1}{2}
\]

when \( n \) is large
Recursion tree

$n$ nodes

Contract a bunch of edges

$\Pr[\text{failure}] = 1/2$

$\frac{n}{\sqrt{2}}$ nodes

Make 2 copies

$\frac{n}{\sqrt{2}}$ nodes

Contract a bunch of edges

$\Pr[\text{failure}] = 1/2$

$\frac{n}{\sqrt{4}}$ nodes

$\frac{n}{\sqrt{4}}$ nodes

Make 2 copies

$\frac{n}{\sqrt{4}}$ nodes

$\frac{n}{\sqrt{4}}$ nodes

Make 2 copies

$\frac{n}{\sqrt{8}}$ nodes

$\frac{n}{\sqrt{8}}$ nodes

$\frac{n}{\sqrt{8}}$ nodes

$\frac{n}{\sqrt{8}}$ nodes

$\frac{n}{2\sqrt{2}}$ nodes

etc.

$\Pr[\text{failure}] = 1/2$
Probability of success

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

Each with probability 1/2
Analysis

• Say the tree has height $d$.

• Let $p_d$ be the probability that there’s a path from the root to a leaf that doesn’t fail.

\[
\begin{align*}
  p_d &= \frac{1}{2} \cdot \Pr \left[ \text{at least one subtree has a successful path} \right] \\
  &= \frac{1}{2} \cdot \left( \Pr \left[ \text{wins} \right] + \Pr \left[ \text{wins} \right] - \Pr \left[ \text{both win} \right] \right) \\
  &= \frac{1}{2} \cdot \left( p_{d-1} + p_{d-1} - p_{d-1}^2 \right) \\
  &= p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2
\end{align*}
\]
Looks like we have 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 $d$, $p_d \geq \frac{1}{d+1}$

Prove this! (Or see hidden slide for a proof).
Recurrence relation

- **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 \( p_{d-1} \geq \frac{1}{d} \).
    - \( p_d = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2 \)
    - \( \geq \frac{1}{d} - \frac{1}{2} \cdot \frac{1}{d^2} \)
    - \( \geq \frac{1}{d} - \frac{1}{d(d+1)} \)
    - \( = \frac{1}{d+1} \)
What does that mean for Karger-Stein?

- For $d = 2\log(n)$
  - that is, $d = \text{the height of the tree}$:

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

- aka,

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

**Claim**: for all $d$, $p_d \geq \frac{1}{d+1}$
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 $1 - \delta$.

- Choosing $\delta = 0.01$ as before, the total runtime is
  $$O(n^2 \log(n) \cdot \log(n)) = O(n^2 \log(n)^2)$$

  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 $\frac{1}{2}$.

• 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!

Before next time

• Pre-lecture exercise: examples of cuts and flows.