Lecture 16

Min Cut and Karger’s Algorithm
Announcements

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

• 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?)
More announcements: Final exam!

• Thursday 3/21, 8:30-11:30am
  • Last name A-LIA: Cubberley Auditorium
  • Last name LIB-Z: Hewlett 200

• Similar format to the midterm
  • (Maybe a bit shorter per time allotted)
  • You are allowed **TWO** pages of cheat sheet.

• Cumulative, but with focus on the second half of the course.
  • Today’s material is fair game but will not be heavily emphasized.
  • Wednesday’s lecture will not have any new material that will be on the final.

• Resources available:
  • Practice Final (up soon)
  • Section this week will be final review
  • Office Hours
  • Slides, CLRS, lecture videos, IPython Notebooks, last year’s lecture notes, ...
How should I study for the final exam?

1. Do practice problems!
   - Practice Exam
   - Sections
   - CLRS/Kleinberg and Tardos/...
   - Google “Practice problems about ____”

2. Do practice problems!

3. Spend some time making your cheat sheet

4. Review the HW – at least one exam question will be very similar to an HW problem.
Piazza Heroes!

Plus everyone who contributes in the comments! (Piazza doesn’t give me a “top” list for that...)
One last announcement

• Course feedback is now open!
• Go to Axess > Student > Course and select “Evaluations.”

• Please fill this out!
• Your feedback is extremely helpful in making this class better going forward!
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 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

• 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.
Different sorts of gambling

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

25

random edge!
Karger’s algorithm

{c,a} {c,b}
{d,a} {d,b}
{e,b}

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

Create a supernode!

{c,a} {c,b}
{d,a} {d,b}
{e,b}

Create a superedge!

{f,e} {f,h}
{g,e} {g,h}

Create a superedge!

Create a supernode!
Karger’s algorithm

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

\[ \{f, e\}, \{f, h\}, \{g, e\}, \{g, h\}, \{e, b\}, \{e, d\} \]
Karger’s algorithm

diagram of a graph with nodes labeled a, b, c, d, e, f, g, h and edges labeled {e,b} {e,d} {f,e} {f,h} {g,e} {g,h}

random edge!
Karger’s algorithm

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

\[ \{f,g\} \{f,e\} \{f,h\} \]
Karger’s algorithm

\begin{align*}
\{e,b\} & \{e,d\} \\
\{f,g\} & \{f,e\} \quad \text{(random edge!)} \\
\{f,h\} & \\
\end{align*}
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 skipped slide with pseudocode

• 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.
Let \( \bar{u} \) denote the SuperNode in \( \Gamma \) containing \( u \)
Say \( E_{\bar{u},\bar{v}} \) is the SuperEdge between \( \bar{u}, \bar{v} \).

### Pseudocode

**Karger( G=(V,E) ):**

- \( \Gamma = \{ \text{SuperNode}(v) : v \in V \} \) // one supernode for each vertex
- \( E_{\bar{u},\bar{v}} = \{(u,v)\} \) for \((u,v)\) in \( E \) // one superedge for each edge
- \( E_{\bar{u},\bar{v}} = \{} \) 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 \( F \)
  - \( \text{merge}(u, v) \) // merge the SuperNode containing \( u \) with the SuperNode containing \( v \).
  - \( F \leftarrow F \setminus E_{\bar{u},\bar{v}} \) // remove all the edges in the SuperEdge between those SuperNodes.

**return** the cut given by the remaining two superNodes.

**merge( u, v ):**

- \( \bar{x} = \text{SuperNode}( \bar{u} \cup \bar{v} ) \) // create a new supernode
- for each \( w \) in \( \Gamma \setminus \{\bar{u}, \bar{v}\} \):
  - \( E_{\bar{x},\bar{w}} = E_{\bar{u},\bar{w}} \cup E_{\bar{v},\bar{w}} \)
  - Remove \( \bar{u} \) and \( \bar{v} \) from \( \Gamma \) and add \( \bar{x} \).

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

The **while** loop runs \( n-2 \) times

**total runtime** \( O(n^2) \)

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

This slide skipped in class
Karger’s algorithm

• Does it work?

• 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, I did it 10,000 times:

The algorithm is only correct about 20% of the time!
That doesn’t sound good

• Too 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?

• 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?

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.

Thing 2: By only contracting edges we are ignoring certain really-not-minimal cuts.
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 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.
- You don’t know what \{a,b,c,d,e\} are.
- Q: What is the minimum of a,b,c,d,e?

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?
This was done 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:

\[ \text{We push the button } t \text{ times and don’t ever get the min} \]

\[ \text{Pr}[ ] = (1 - 0.2)^t \]

\[ \text{We push the button 5 times and don’t ever get the min} \]

\[ \text{Pr}[ ] = (1 - 0.2)^5 \approx 0.33 \]

\[ \text{We push the button 10 times and don’t ever get the min} \]

\[ \text{Pr}[ ] = (1 - 0.2)^{10} \approx 0.1 \]
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! (with probability about 0.66)
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 something?

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 most 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 \( \frac{1}{\binom{n}{2}} \)

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

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

   According to the claim, at least \( \frac{1}{\binom{n}{2}} \)

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

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

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

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

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

   According to the claim, at least \( \frac{1}{\binom{n}{2}} \)

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

   \( \binom{n}{2} \ln \left( \frac{1}{\delta} \right) \) times.
Theorem
Assuming the claim about $1/\binom{n}{2}$ ...

• Suppose $G$ has $n$ vertices.

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

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

How many repetitions would you need if instead of Karger we just chose a uniformly random cut?
What’s the running time?

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

Treating \( \delta \) 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?

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

\[ \text{at least } \frac{1}{\binom{n}{2}} \]
Now let’s prove that claim
Say that $S^*$ is a minimum cut.

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

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

These two edges haven’t been chosen for contraction!
Focus in on:
\[ \Pr[ e_j \text{ doesn’t cross } S^* \mid e_1, \ldots, e_{j-1} \text{ don’t cross } S^* ] \]

- 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 supernode has at least \( k \) (original) edges coming out.
    - Otherwise we’d have a smaller cut.
  - Thus, there are at least \( (n-j+1)k/2 \) edges total.
    - \( b/c \) there are \( n - j + 1 \) supernodes left, each with \( k \) edges.

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

\[
\frac{k}{\binom{(n-j+1)k}{2}} = \frac{2}{n-j+1}
\]
Focus in on:
\[ \text{PR} [ \text{e}_j \text{ doesn’t cross S* } | \text{ e}_1, \ldots, \text{e}_{j-1} \text{ don’t cross S* } ] \]

- So the probability that we choose one of the k edges crossing S* at step j is at most:
  \[
  \frac{k}{\binom{(n-j+1)k}{2}} = \frac{2}{n-j+1}
  \]

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

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

\[
= \left( \frac{n-2}{n} \right) \left( \frac{n-3}{n-1} \right) \left( \frac{n-4}{n-2} \right) \left( \frac{n-5}{n-3} \right) \left( \frac{n-6}{n-4} \right) \ldots \left( \frac{4}{6} \right) \left( \frac{3}{5} \right) \left( \frac{2}{4} \right) \left( \frac{1}{3} \right)
\]
Now let’s prove that claim

Say that $S^*$ is a minimum cut.

• Suppose the edges that we choose are $e_1, e_2, \ldots, e_{n-2}$

$\text{PR}[ \text{return } S^* ] = \text{PR}[ \text{none of the } e_i \text{ cross } S^* ]$

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

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

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

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.

• 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.

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

• If we have a Monte-Carlo algorithm with a small success probability,
• and we can check how good a solution is,
• Then 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

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

Probability that we didn’t mess up: 12/14
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: \( \frac{11}{13} \)

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!

Create a supernode!
Karger’s algorithm

Create a supernode!

Create a superedge!

Create a superedge!

Create a supernode!
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.

random edge!
Karger’s algorithm
Karger’s algorithm

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

Probability that we didn’t mess up: \(\frac{9}{11}\)
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:
\[ \frac{3}{5} \]
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...

This branch made a bad choice.  

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

FORK!
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 \( \left( \frac{n}{\sqrt{2}} \right) \frac{\sqrt{2}}{} = \frac{n}{2} \) 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

Contract a bunch of edges

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

Make 2 copies

Contract a bunch of edges

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

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

Make 2 copies

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

Contract a bunch of edges

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

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

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

$\frac{n}{\sqrt{8}}$ nodes
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...
  $$T(n) = O(n^2\log(n))$$
Two questions

• Does this work?

• Is it fast?
  • Yes, $O(n^2 \log(n))$. 
**Why \( \frac{n}{\sqrt{2}} \) ?**

- Suppose the first \( n-t \) edges that we choose are \( e_1, e_2, \ldots, e_{n-t} \)
- \( \text{PR} \) [none of \( e_1, e_2, \ldots, e_{n-t} \) cross \( S^* \)]
  
  \[= \text{PR} \ [ \text{e}_1 \text{ doesn’t cross } S^* ] \]
  \[\times \text{PR} \ [ \text{e}_2 \text{ doesn’t cross } S^* \mid \text{e}_1 \text{ doesn’t cross } S^* ] \]
  \[\vdots \]
  \[\times \text{PR} \ [ \text{e}_{n-t} \text{ doesn’t cross } S^* \mid \text{e}_1,\ldots,\text{e}_{n-t-1} \text{ don’t cross } S^* ] \]

Suppose we contract \( n – t \) edges, until there are \( t \) supernodes remaining.
Why $n/\sqrt{2}$?

- Suppose the first $n-t$ edges that we choose are $e_1, e_2, \ldots, e_{n-t}$
- **PR** [none of $e_1, e_2, \ldots, e_{n-t}$ cross $S^*$]

\[
\begin{align*}
&= \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \left(\frac{n-4}{n-2}\right) \left(\frac{n-5}{n-3}\right) \left(\frac{n-6}{n-4}\right) \cdots \left(\frac{t+1}{t+3}\right) \left(\frac{t}{t+2}\right) \left(\frac{t-1}{t+1}\right) \\
&= \frac{t \cdot (t-1)}{n \cdot (n-1)} \\
&= \frac{n}{\sqrt{2}} \cdot \left(\frac{n}{\sqrt{2}} - 1\right) \\
&\approx \frac{1}{2}
\end{align*}
\]

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

When $n$ is large

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

- Start with $n$ nodes.
- Contract a bunch of edges to get $\frac{n}{\sqrt{2}}$ nodes.
- Make 2 copies of the remaining $\frac{n}{\sqrt{2}}$ nodes.
- Repeat the process with $\frac{n}{\sqrt{2}}$ nodes.

- Pr[ failure ] = 1/2

- The recursion continues with smaller and smaller sub-trees until the base case is reached.

- The probability of failure at each step is 1/2.

- The recursion tree continues to expand with each level having $\frac{n}{\sqrt{2}}$ nodes.

- The tree structure and the process of contracting and copying edges continues until the base case is reached.

- The recursion tree is a visual representation of the recursive process.
Probability of success

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

Each with probability $1/2$
The problem we need to analyze

• Let T be binary tree of depth $2\log(n)$
• 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 $p_d$ be the probability that there’s a path from the root to a leaf that doesn’t fail.

\[ p_d = \frac{1}{2} \cdot \Pr \left[ \text{at least one subtree has a successful path} \right] \]

\[ = \frac{1}{2} \cdot \left( \Pr \left[ \text{wins} \right] + \Pr \left[ \text{wins} \right] \right) \]

\[ = \frac{1}{2} \cdot \left( \Pr \left[ \text{wins} \right] + \Pr \left[ \text{both win} \right] \right) \]

\[ = \frac{1}{2} \cdot \left( p_{d-1} + p_{d-1} - p_{d-1}^2 \right) \]

\[ = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2 \]
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 $d$, $p_d \geq \frac{1}{d+1}$

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 \( p_{d-1} \geq \frac{1}{d} \).
    
    • \( p_d = p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2 \)
    
    • \( \geq \frac{1}{d} - \frac{1}{2} \cdot \frac{1}{d} \)
    
    • \( \geq \frac{1}{d} - \frac{1}{d(d+1)} \)
    
    • \( = \frac{1}{d+1} \)

\[ \begin{align*}
p_d &= p_{d-1} - \frac{1}{2} \cdot p_{d-1}^2 \\
p_0 &= 1
\end{align*} \]
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 $1 - \delta$.

• 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
  $$O(n^2 \log(n) \cdot \log(n)) = O(n^2 \log^2(n))$$

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 (probably) 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

• Recap of what we’ve done this quarter
• What’s next???

• Reminder: Please complete course evaluations!