Parallel And Sequential Data Structures and Algorithms

Graph Contraction, Part I

Announcements

- **ShortLab is already out, and due next Monday, Apr. 6**
- **MT2 Grades are released, Solution Session on Friday during Lecture time**
- **Group Session for ShortLab in Wednesday and Thursday OH!**

Learning Objectives

- Introduce **Graph Contraction**, a way to achieve polylogarithmic span algorithms on graphs
- Define two different types of Graph Contraction: **Edge Contraction** and **Star Contraction**, and analyze their performance
- Present implementation details for Star Contraction

What is Graph Contraction?

What is Graph Contraction?

- Goal: **get polylogarithmic span** algorithms on the size of the graph (for any graph).
- *Note: we have not achieved this for any graph problems discussed so far in 15-210.*

What is Graph Contraction?

- Goal: **get polylogarithmic span** algorithms on the size of the graph (for any graph).
- *Note: we have not achieved this for any graph problems discussed so far in 15-210.*
- Ex: BFS is parallel, but the span has a term that was the diameter of the graph.

Theorem (Cost of Parallel BFS): on a simple graph, both imperative and functional parallel BFS cost $O\left(D \log^2(n)\right)$ span.

What is Graph Contraction?

- Goal: **get polylogarithmic span** algorithms on the size of the graph (for any graph).
- *Note: we have not achieved this for any graph problems discussed so far in 15-210.*
- Ex: BFS is parallel, but the span has a term that was the diameter of the graph.
- Similar story for Bellman-Ford and Floyd-Warshall

Applications of Graph Contraction

- Minimum Spanning Trees (MSTs),
  foreshadowing… 2 lectures from now!

Definition (Tree): a graph with *n* vertices that satisfies two of the following properties, (1) acyclic, (2) contains *n – 1* edges, (3) connected graph. *Knowing any pair lets you derive the third – see 15-251!* ☺

Definition (Spanning Tree): for a graph with *n* vertices, a spanning tree is a subgraph that connects all *n* vertices. A **Minimum Spanning Tree** is the lowest cost spanning tree for a weighted graph.

Applications of Graph Contraction

- Minimum Spanning Trees (MSTs),
  foreshadowing… 2 lectures from now!
- Check if a graph is bipartite

Definition (Bipartite): a graph $G = (V, E)$ is bipartite if V can be partitioned into two sets X and Y s.t. for all edges (u, v) in E, exactly one of u and v is in X, and therefore the other one is in Y.

Applications of Graph Contraction

- Minimum Spanning Trees (MSTs),
  foreshadowing… 2 lectures from now!
- Check if a graph is bipartite
- Find undirected bi-connected components

Definition (Bi-Connected Components): a subgraph of a graph *G* s.t. if any vertex is removed, the graph remains connected.

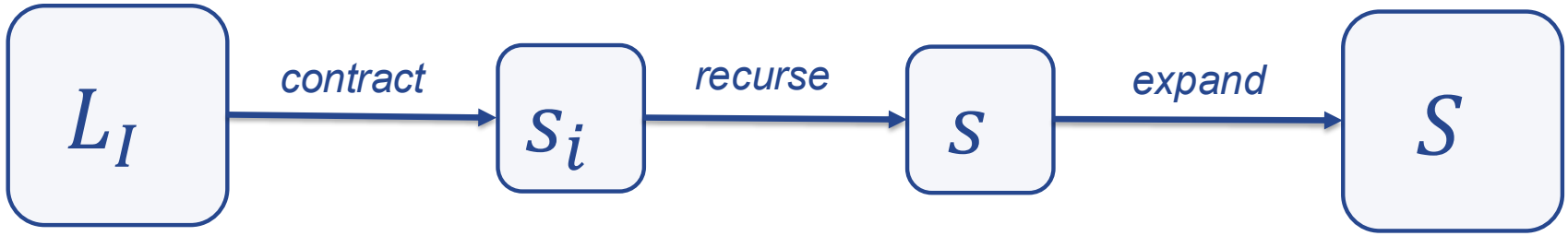
Applications of Graph Contraction

- Minimum Spanning Trees (MSTs), *foreshadowing… 2 lectures from now!*
- Check if a graph is bipartite
- Find undirected bi-connected components
- Min Cut in a graph

Definition (Cut): a partition of a graph's vertices into S and $V - S$. A **Min Cut** of a graph G is a cut that minimizes the sum of the weights of the edges that span the cut, i.e., edges (u, v) s.t. exactly one of u and v is in S, and the other is not in S.

The Concept of Contraction

Idea (Contraction): to solve a large instance of a problem L_I, first shrink it to a smaller instance s_i that we recursively solve to give a solution *s*. Then, we can expand the solution *s* to construct the overall solution *S* to the larger instance.



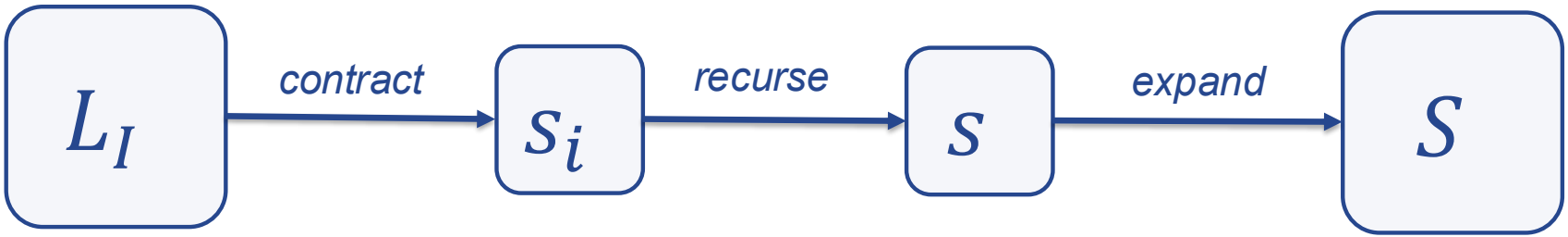
- If we can shrink the problem geometrically, then the number of levels is guaranteed to be logarithmic.

The Concept of Contraction

- It's basically a recurrence of the form

$$Span(n) = Span(\alpha n) + O\big(S(n)\big)$$

- Where $0 < \alpha < 1$. This leads to an $O(S(n)\log(n))$ span algorithm.



- If we can shrink the problem geometrically, then the number of levels is guaranteed to be logarithmic.

The Concept of Contraction

- It's basically a recurrence of the form

$$Span(n) = Span(\alpha n) + O\big(S(n)\big)$$

- Where $0 < \alpha < 1$. This leads to an $O(S(n)\log(n))$ span algorithm.
- **You've seen this recurrence before! Where??????**

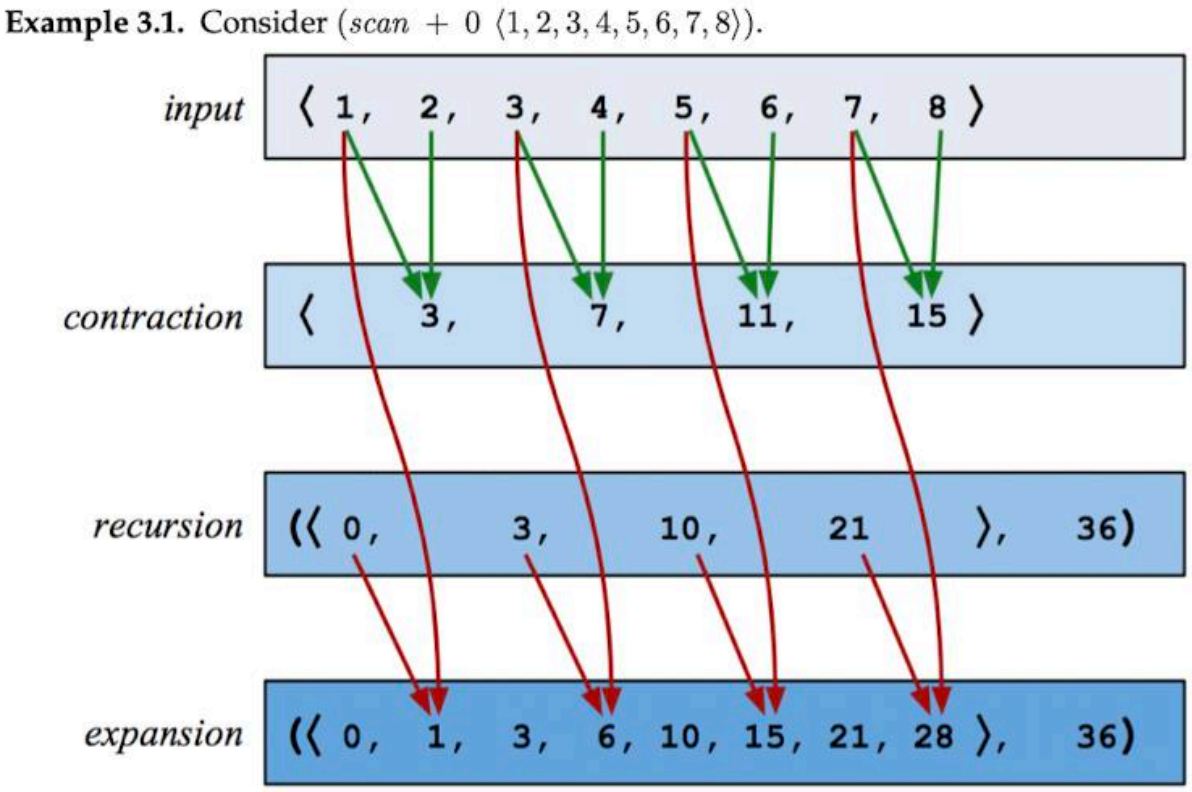
The Concept of Contraction

- It's basically a recurrence of the form

$$Span(n) = Span(\alpha n) + O\big(S(n)\big)$$

- Where $0 < \alpha < 1$. This leads to an $O(S(n)\log(n))$ span algorithm.
- **You've seen this recurrence before! Where??????**
- **In Week 3, the $O(\log(n))$ span implementation of Scan!**

The Concept of Contraction



Keep these three steps in mind:

1. *Contraction*
2. *Recursion*
3. *Expansion*

Types of Graph Contraction

Towards a concrete algorithm...

- The type of contraction used should be tailored to the problem being solved.

Towards a concrete algorithm...

- The type of contraction used should be tailored to the problem being solved.
- As a running example, we will analyze the connectivity of undirected graphs *(i.e., count the number of components, and identify which component each vertex is in)*
- Therefore, the goal of our contraction is to
  (1) shrink the graph, and
  (2) maintain connectivity.

Towards a concrete algorithm…

- Running example: analyze the connectivity of undirected graphs *(i.e., count the number of components, and identify which component each vertex is in)*
- Goals: (1) shrink the graph and (2) maintain connectivity

Towards a concrete algorithm…

- Running example: analyze the connectivity of undirected graphs *(i.e., count the number of components, and identify which component each vertex is in)*
- Goals: (1) shrink the graph and (2) maintain connectivity

Algorithm (Contraction):

- Take a connected set of edges and vertices, and contract them into a new **"super vertex"**
- Can do this as many times **in parallel** as long as the vertices in each set are **disjoint**.
- Edges between super vertices are computed.

Edge Contraction

Edge Contraction

- The “connected sets of edges and vertices” is just one edge and two vertices, or a single vertex and no edges

Algorithm (Contraction):

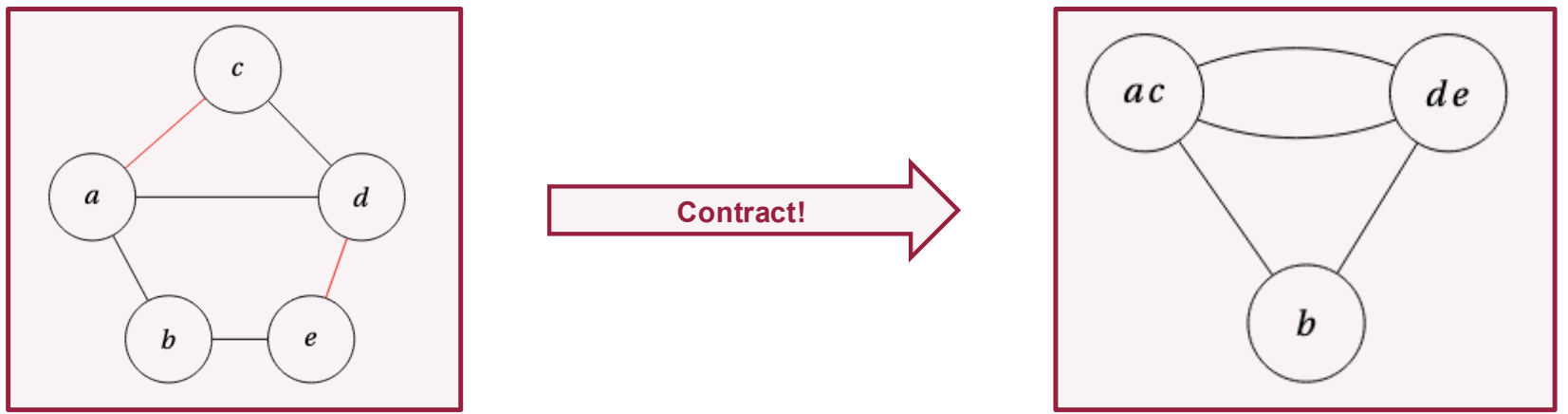
- Take a connected set of edges and vertices, and contract them into a new **“super vertex”**
- Can do this as many times **in parallel** as long as the vertices in each set are **disjoint**.
- Edges between super vertices are computed.

Edge Contraction

- The “connected sets of edges and vertices” is just one edge and two vertices, or a single vertex and no edges

Algorithm (Contraction):

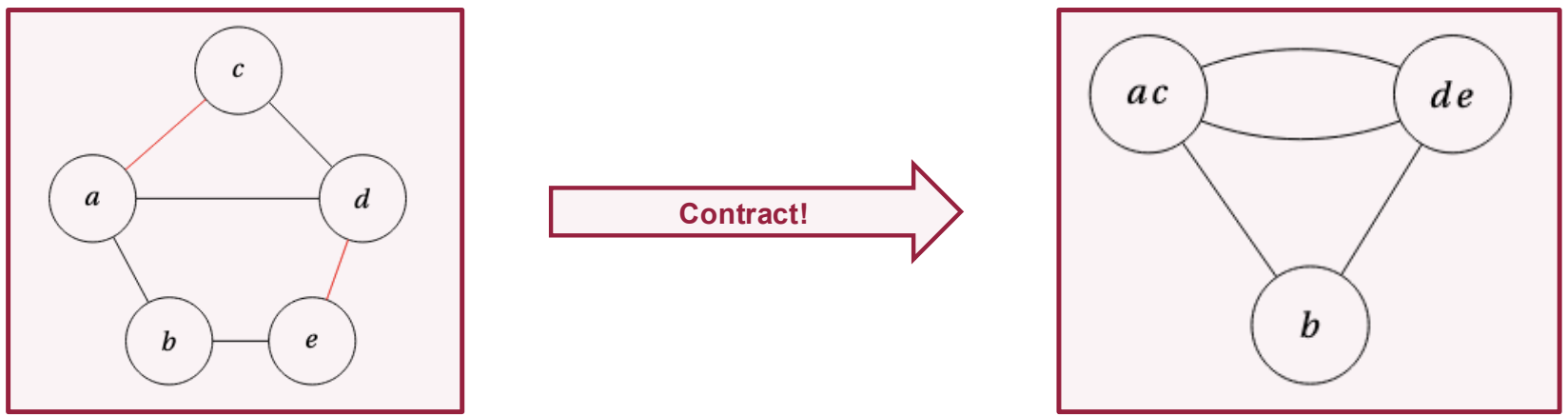
- Take a connected set of edges and vertices, and contract them into a new **“super vertex”**
- Can do this as many times **in parallel** as long as the vertices in each set are **disjoint**.
- Edges between super vertices are computed.



Edge Contraction: Which edges to pick?

- Want to pick as many as possible, and their endpoints must be disjoint...

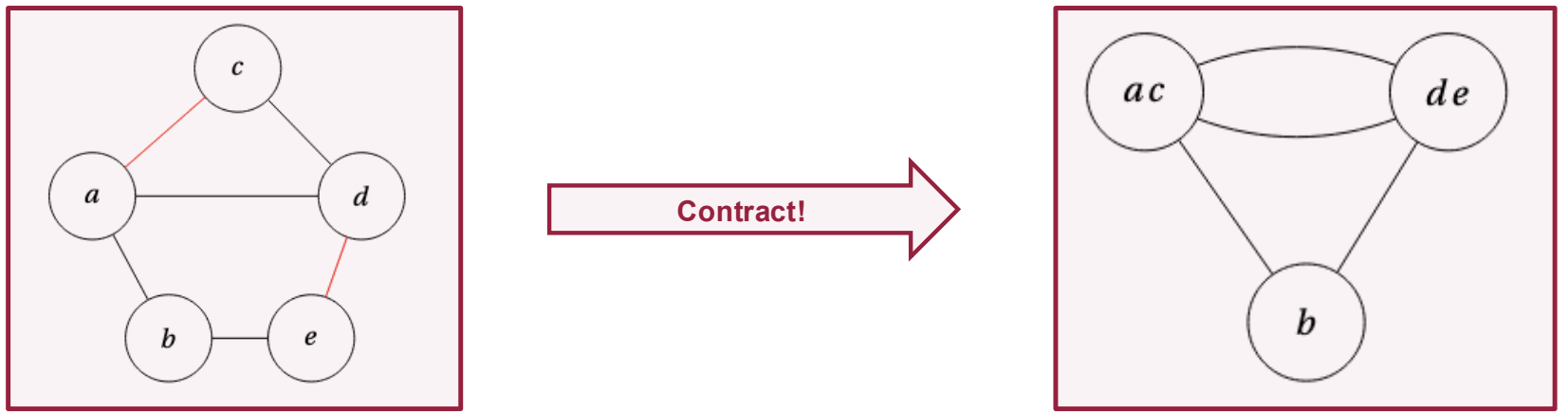
what is that called?



Edge Contraction: Which edges to pick?

- Want to pick as many as possible, and their endpoints must be disjoint… So find a **matching**!

Definition (Matching): a subset of edges whose endpoints are disjoint.



Edge Contraction: Which edges to pick?

- Want to pick as many as possible, and their endpoints must be disjoint… So find a **matching**!

Definition (Matching): a subset of edges whose endpoints are disjoint.

- There are algorithms for computing maximum matchings in a graph (*again, see 15-251 ☺),* but we want something that is **faster** and **very parallel.** How…

Edge Contraction: Which edges to pick?

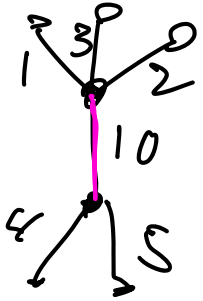
- Want to pick as many as possible, and their endpoints must be disjoint… So find a **matching**!

Definition (Matching): a subset of edges whose endpoints are disjoint.

- There are algorithms for computing maximum matchings in a graph (*again, see 15-251 ☺)*, but we want something that is **faster** and **very parallel.** How… Use **randomness**! But how…

Edge Contraction: Random Matching

- Suppose we label each edge with a random priority. How could we use the priority to select which edges to contract?



Edge Contraction: Random Matching

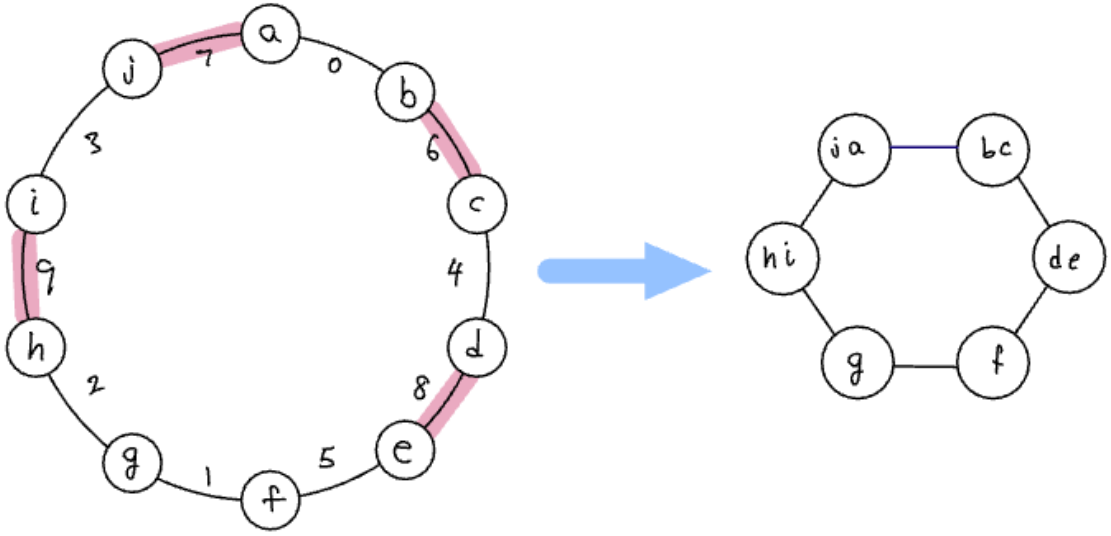
- Suppose we label each edge with a random priority. How could we use the priority to select which edges to contract?

Algorithm (Edge Contraction): given each edge has random priority, select an edge for contraction if its priority is higher than all its incident edges.

Edge Contraction: Random Matching

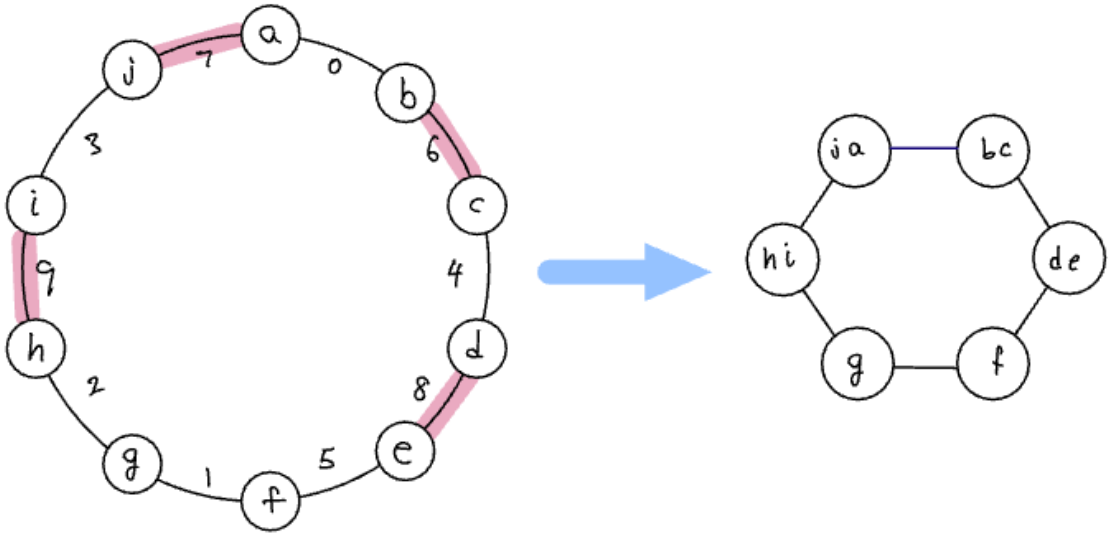
Algorithm (Edge Contraction): given each edge has random priority, select an edge for contraction if its priority is higher than all its incident edges.

- *Notice:*
  for a cycle,
  no matter which
  edges are
  contracted,
  the resulting graph
  is still a cycle.



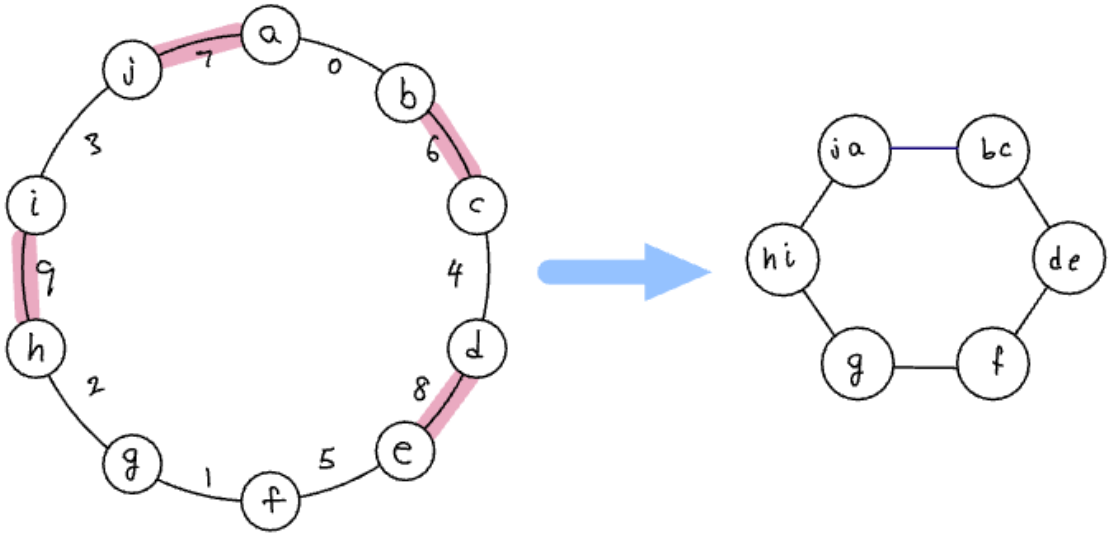
Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?



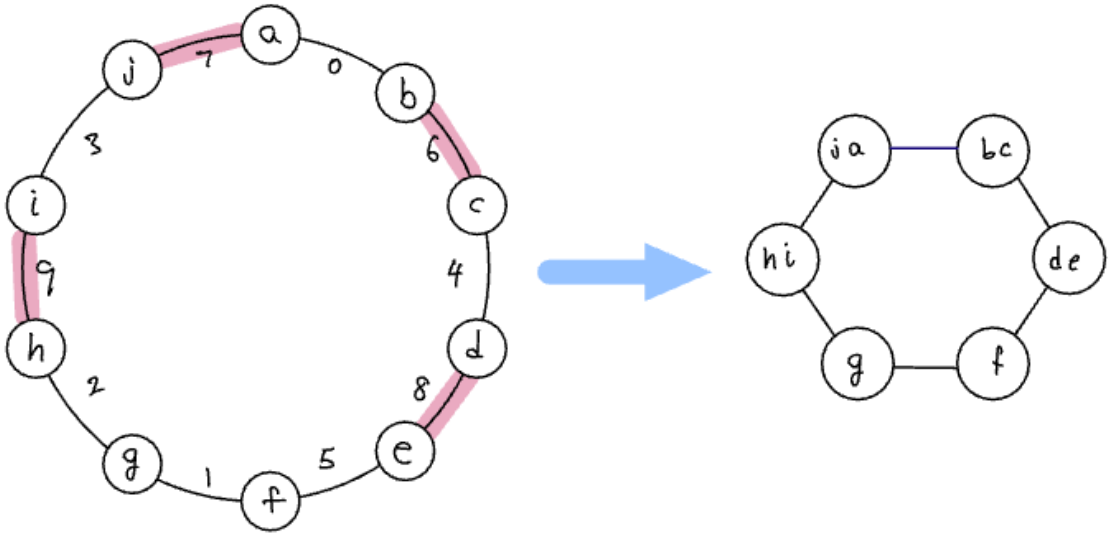
Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?
- Ans: $\frac{1}{3}$



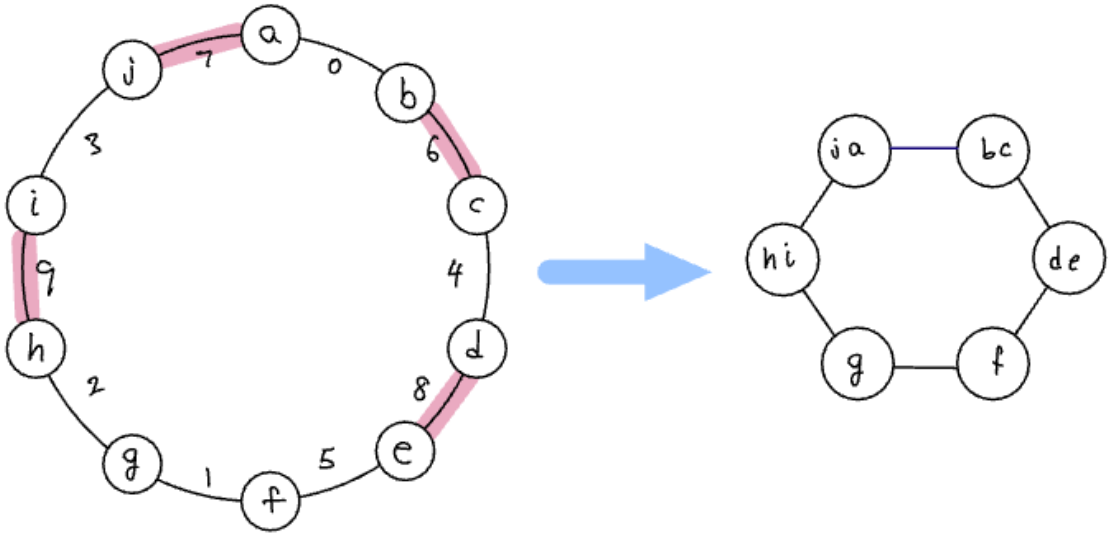
Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?
- Ans: $\frac{1}{3}$
- There are n edges in C_n. What is the expected number of edges selected for contraction?



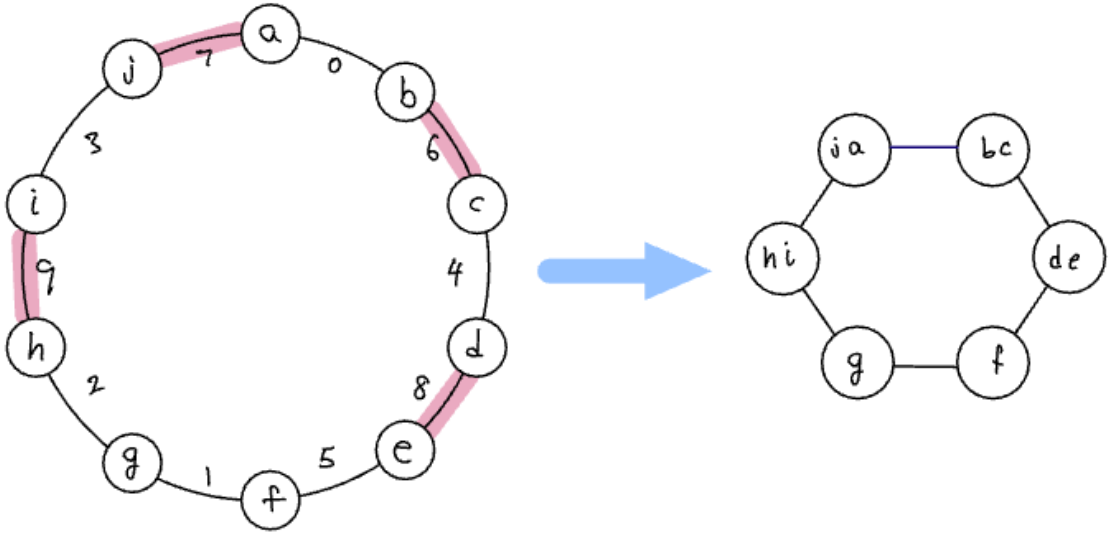
Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?
- Ans: $\frac{1}{3}$
- There are n edges in C_n. What is the expected number of edges selected for contraction?
- Ans: $\frac{1}{3}n$



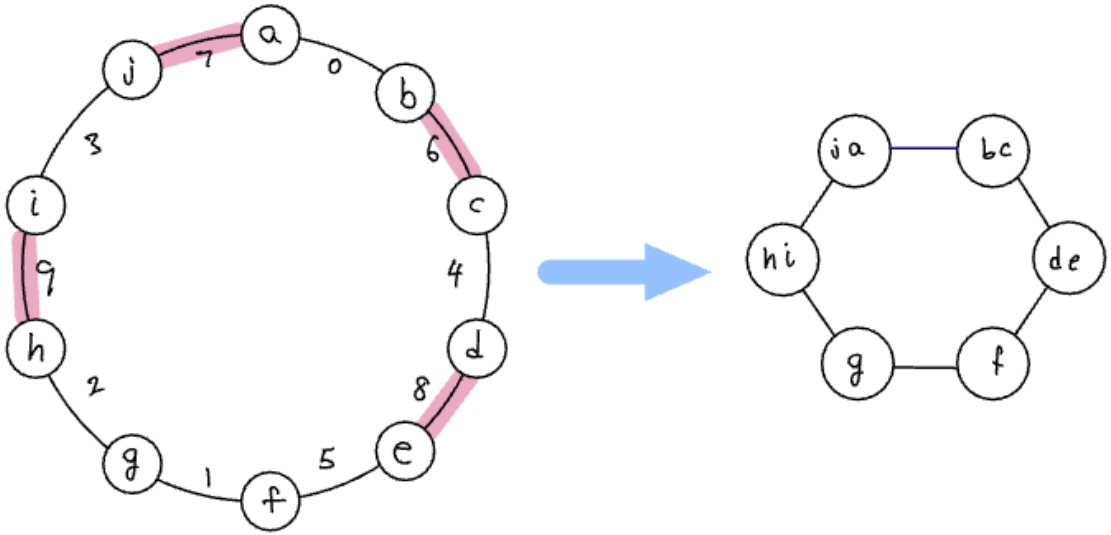
Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?
- Ans: $\frac{1}{3}$
- There are n edges in C_n. What is the expected number of edges ***remaining after contraction?***



Edge Contraction: Cost Analysis

- Consider a graph C_n, a cycle with n vertices. What is the probability that an edge is selected for contraction?
- Ans: $\frac{1}{3}$
- There are n edges in C_n. What is the expected number of edges ***remaining after contraction?***
- Ans: $\frac{2}{3}n$



Edge Contraction: Cost Analysis

- Consider C_n, a cycle with n vertices. There are n edges in C_n.
  - $\Pr[\text{contract any given edge}] = \frac{1}{3}$
  - $\mathbb{E}[\#\text{ edges contracted}] = \frac{1}{3}n$
  - $\mathbb{E}[\#\text{ edges remaining}] = \frac{2}{3}n$
- How many times will we have to apply edge contraction in order to contract to a graph of just one vertex?

Edge Contraction: Cost Analysis

- Consider C_n, a cycle with *n* vertices. There are *n* edges in C_n.
  - $\Pr[\text{contract any given edge}] = \frac{1}{3}$
  - $\mathbb{E}[\#\text{ edges contracted}] = \frac{1}{3}n$
  - $\mathbb{E}[\#\text{ edges remaining}] = \frac{2}{3}n$
- How many times will we have to apply edge contraction in order to contract to a graph of just one vertex?
- Ans: $O(\log n)$ w.h.p. *(remember Skittles Lemma?)*

Edge Contraction: Uh oh…

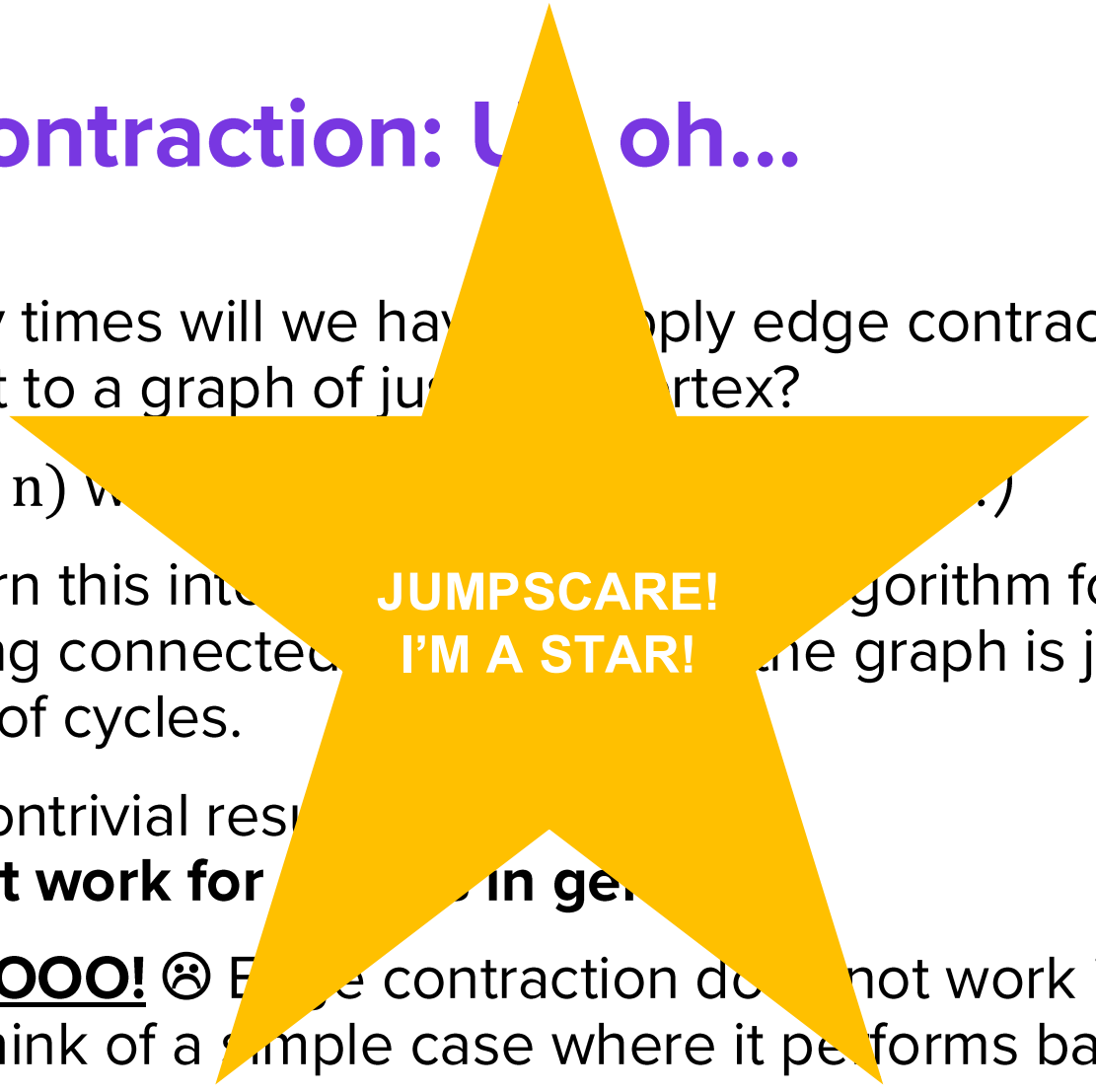
- How many times will we have to apply edge contraction in order to contract to a graph of just one vertex?
- Ans: $O(\log n)$ w.h.p. *(remember Skittles Lemma?)*
- We can turn this into an efficient parallel algorithm for determining connected components if the graph is just a collection of cycles.
- That's a nontrivial result!
  But does it work for graphs in general?

Edge Contraction: Uh oh…

- How many times will we have to apply edge contraction in order to contract to a graph of just one vertex?
- Ans: $O(\log n)$ w.h.p. *(remember Skittles Lemma?)*
- We can turn this into an efficient parallel algorithm for determining connected components if the graph is just a collection of cycles.
- That's a nontrivial result!
  But does it work for graphs in general?
- Ans: **NOOOOO!** ☹ Edge contraction does not work in general. Can you think of a simple case where it performs badly?

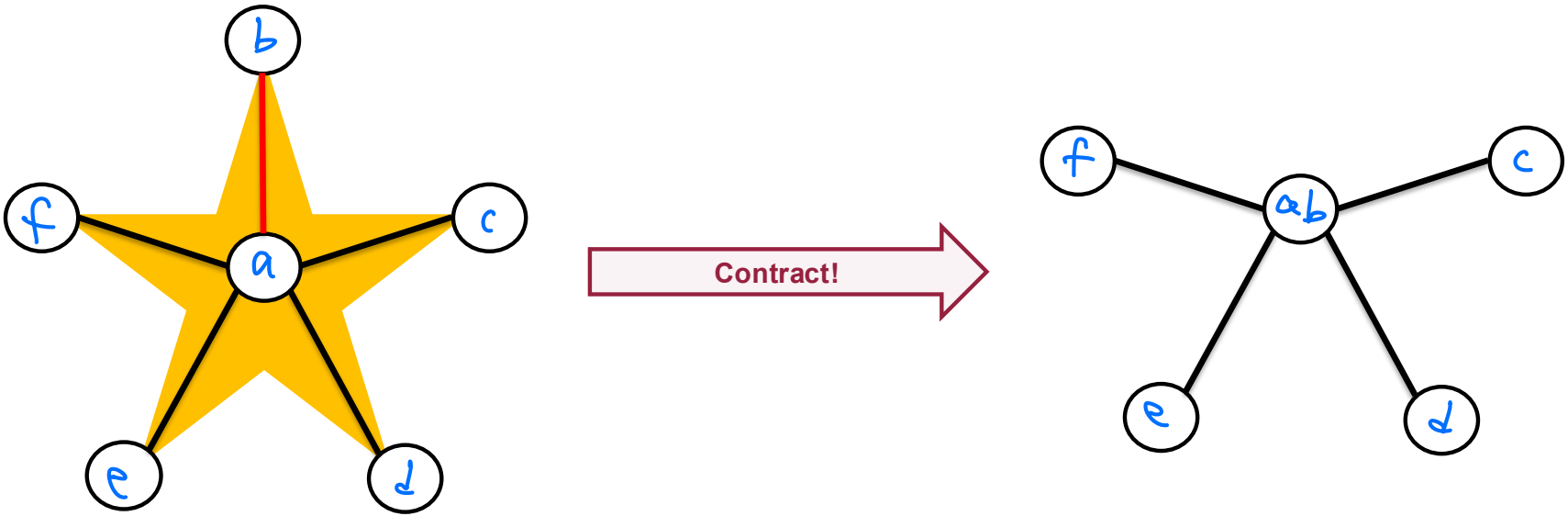
Edge Contraction: U oh…

- How many times will we hav pply edge contraction in order to contract to a graph of ju rtex?
- Ans: $O(\log n)$ w .)
- We can turn this int gorithm for determining connected he graph is just a collection of cycles.
- That's a nontrivial res **But does it work for in ge**
- Ans: **NOOOOO!** ☹ E e contraction do not work in general. Can you think of a simple case where it pe forms badly?



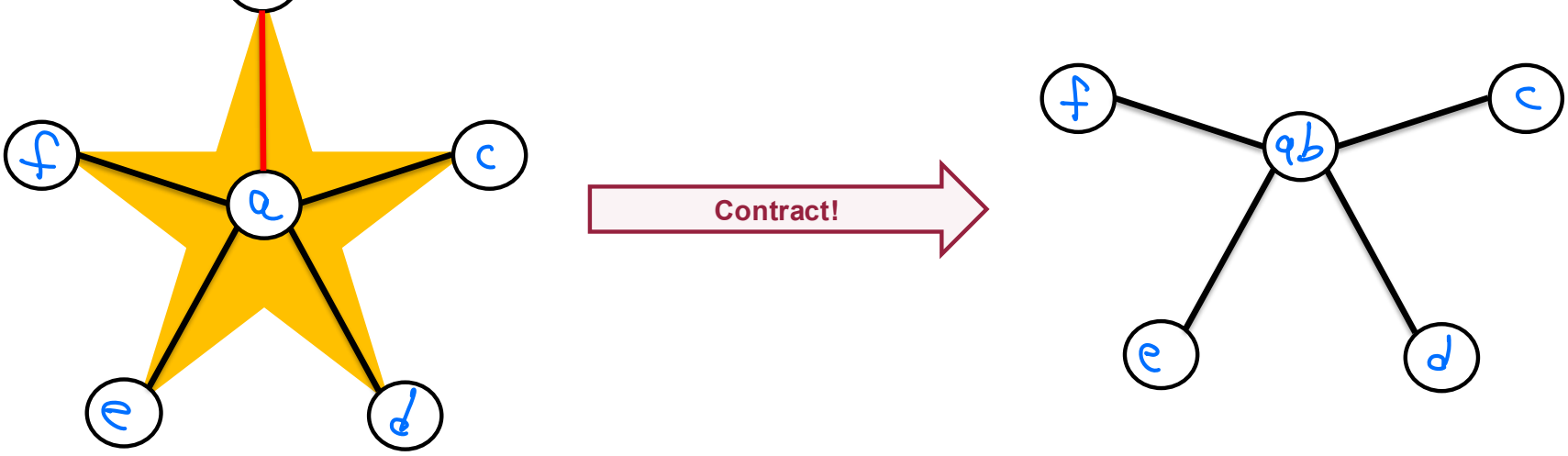
Edge Contraction on Star Graph

- How about a star graph?
- How many iterations do we need now?



Edge Contraction on Star Graph

- How about a star graph?
- How many iterations do we need now?
  Ans: ***n – 1*** iterations are required to reduce a star with one center b and *n – 1* satellites.



Star Contraction

Star Contraction

- So edge contraction does not always work ☹
- But maybe this hard star case can lead us to a better way!

Star Contraction

- So edge contraction does not always work ☹
- But maybe this hard star case can lead us to a better way!
- Suppose we are allowed to contract a node ***and*** some of its neighbors. A little “star-shaped” set of nodes.

Star Contraction

- So edge contraction does not always work ☹
- But maybe this hard star case can lead us to a better way!
- Suppose we are allowed to contract a node ***and*** some of its neighbors. A little “star-shaped” set of nodes.
- We can make this work!!!

Star Contraction: Yay Randomness!

- Again, randomness comes to the rescue to create a really simple and effective algorithm for finding stars to contract.

Star Contraction: Yay Randomness!

- Again, randomness comes to the rescue to create a really simple and effective algorithm for finding stars to contract.

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a “star center.” If it comes up tails, then connect it to **any** neighboring star center!

Star Contraction: Yay Randomness!

- Again, randomness comes to the rescue to create a really simple and effective algorithm for finding stars to contract.

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to **any** neighboring star center!

- Wait. What if it doesn't have a neighboring star center? What is a simple thing we could do?

Star Contraction: Yay Randomness!

- Again, randomness comes to the rescue to create a really simple and effective algorithm for finding stars to contract.

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to **any** neighboring star center!

- Wait. What if it doesn't have a neighboring star center? What is a simple thing we could do?
- Ans: Just make it a star center of a "degenerate star" with no satellites! So it just sticks around as a vertex in the contracted graph. ***Could such a simple hack possibly work????***

Star Contraction Lemma

Star Contraction Lemma: suppose that star contraction is applied to a graph with n_1 non-isolated vertices and n_2 isolated vertices. The expected number of vertices after star contraction is at most $\frac{3}{4}n_1 + n_2$.

Star Contraction Lemma

Star Contraction Lemma: suppose that star contraction is applied to a graph with n_1 non-isolated vertices and n_2 isolated vertices. The expected number of vertices after star contraction is at most $\frac{3}{4}n_1 + n_2$.

- *Proof:* We're only concerned with non-isolated vertices. A non-isolated vertex *v* is eliminated if it flips tails and at least one of its neighbors flips heads.

Star Contraction Lemma

Star Contraction Lemma: suppose that star contraction is applied to a graph with n_1 non-isolated vertices and n_2 isolated vertices. The expected number of vertices after star contraction is at most $\frac{3}{4}n_1 + n_2$.

- *Proof:* We're only concerned with non-isolated vertices. A non-isolated vertex *v* is eliminated if it flips tails and at least one of its neighbors flips heads.

- By virtue of being "non-isolated", *v* has at least one neighbor. $\Pr[\text{neighbor flips heads}] = \frac{1}{2}$. So the probability *v* is contracted is at least $\frac{1}{4}$ – if it has more neighbors, this probability is higher.

Star Contraction Lemma

Star Contraction Lemma: suppose that star contraction is applied to a graph with n_1 non-isolated vertices and n_2 isolated vertices. The expected number of vertices after star contraction is at most $\frac{3}{4}n_1 + n_2$.

- *Proof:* We're only concerned with non-isolated vertices. A non-isolated vertex *v* is eliminated if it flips tails and at least one of its neighbors flips heads.

- By virtue of being "non-isolated", *v* has at least one neighbor. $\Pr[\text{neighbor flips heads}] = \frac{1}{2}$. So the probability *v* is contracted is at least $\frac{1}{4}$ – if it has more neighbors, this probability is higher.

- The Lemma follows immediately. Q.E.D. ☺

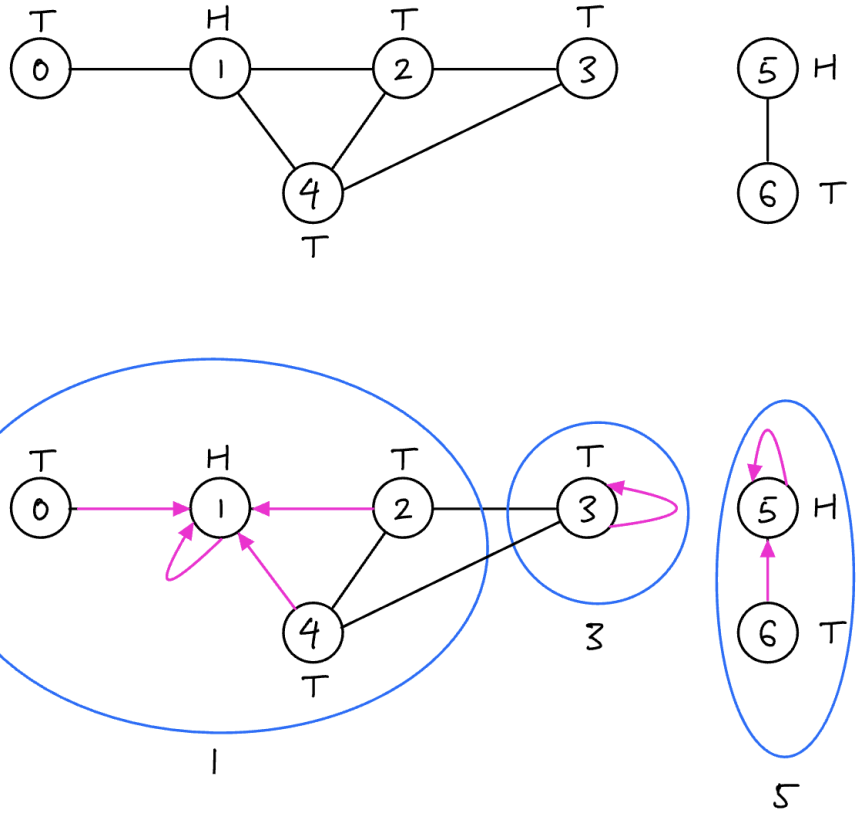
Star Contraction: Cost Analysis

- By the Skittles Lemma, it also follows that the expected number of rounds of star contraction needed until all vertices are isolated is $O(\log n)$ w.h.p.

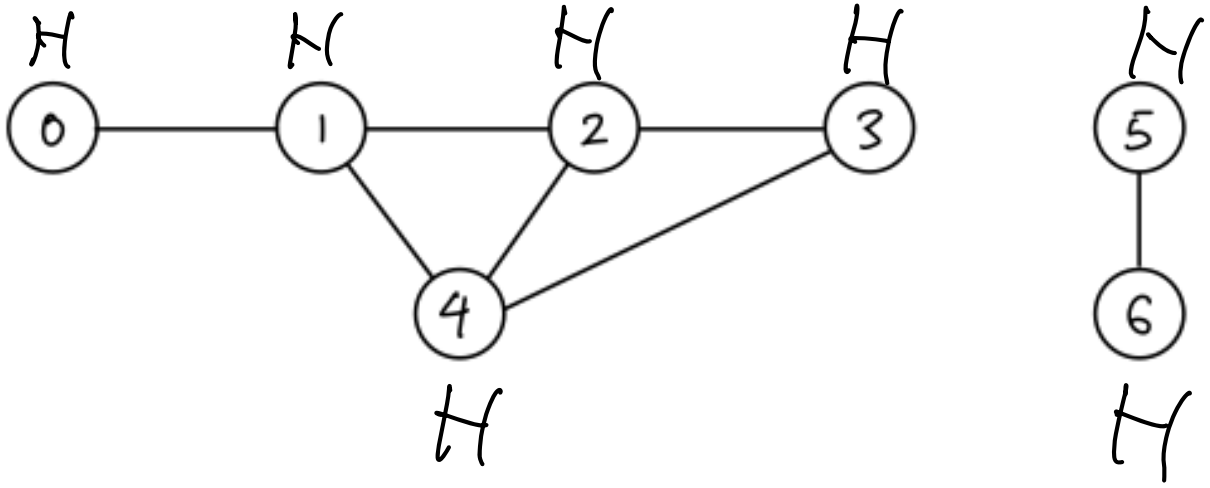
- And we've made no assumptions about the structure of the graph!

Star Contraction Example 1

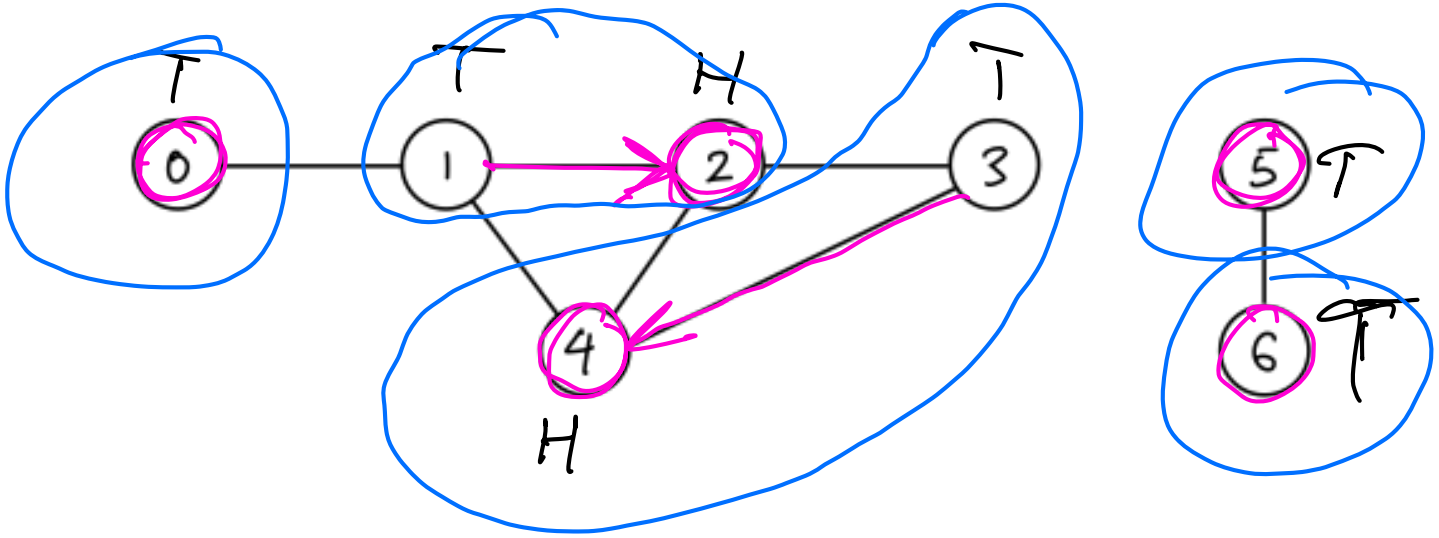
- Given a 7-vertex graph with two components, flip the coins on each vertex.
- Now that we have our star partitions,
- Contract them!



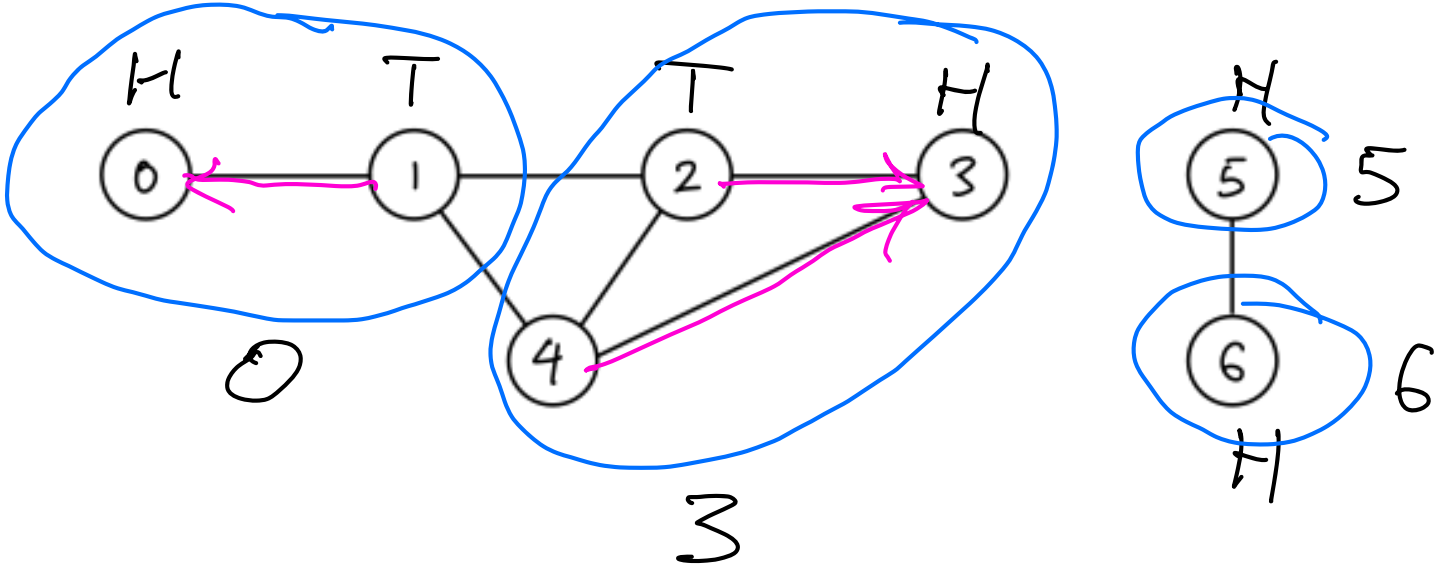
Star Contraction Example 2



Star Contraction Example 3



Star Contraction Example 4



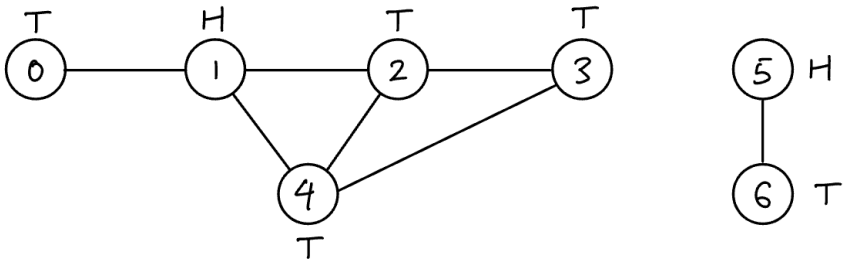
Star Contraction Example 1 (contd)

The input in this example is:

V = [0, 1, 2, 3, 4, 5, 6]

E = [(0, 1), (1, 0), (1, 2), (2, 1), 2, 3), (3, 2), (1, 4), (4, 1),
(4, 2), (2, 4), (4, 3), (3, 4), (5, 6), (6, 5)]

n = 7

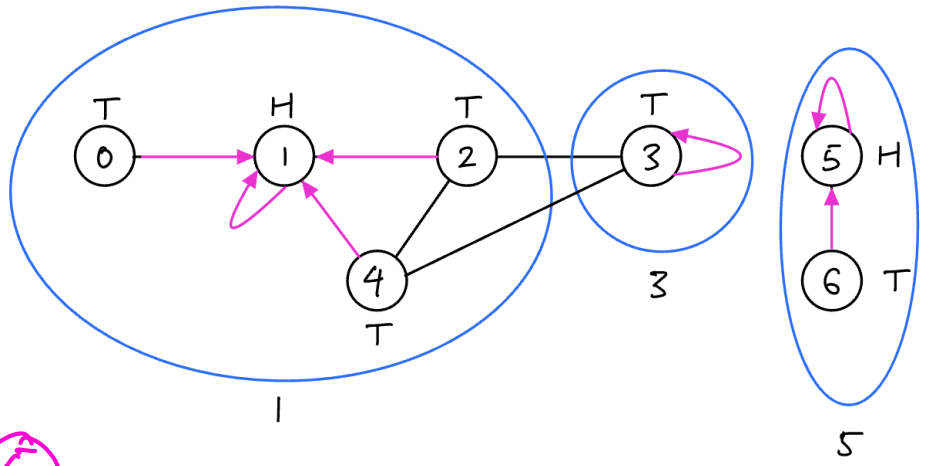


The output in this example is:

Vc = [1, 3, 5]

Ec = [(1, 3), (3, 1), (1, 3), (3, 1)]

P = [1, 1, 1, 3, 1, 5, 5]



Star Partition Implementation

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to a neighboring star center!

The input to the algorithm

- (V, E, n). These parameters represent a graph.
- The vertices are numbered range $[0, n-1]$.
- V is a sequence containing the vertices, which are a subset of $\{0, \ldots, n-1\}$.
- The edges are represented by E:
  This is a sequence of pairs (u, v) where u and v are vertices.
  There are no self-loops, and if the edge (u, v) occurs in E then so does (v, u)

The algorithm returns

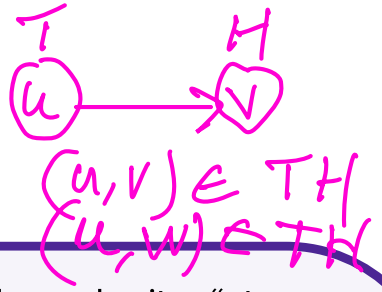
- V_c, the vertices that become star centers,
- E_c, the edges that remain between the star centers,
- P, a sequence which maps each vertex of V to the star center with which it is associated.

Star Partition Implementation

Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to a neighboring star center!

```
fun starPartition (V, E, n):
(1) Generate heads, a sequence of n random Boolean values that are true w.p. 1/2

(2) Initialize a sequence of parent values P = [0, 1, …, n–1]. Each vertex is initially its
own parent.

(3) For every vertex u with heads[u]=false, if it has an edge (u,v) where heads[v]=true then
P[u]← v. (If there is more than one such v then pick arbitrarily.)

(4) Compute the center vertices Vc, represented as a sequence of centers, namely the
vertices for which P[v] = v.

(5) Compute the edges Ec of the star partitioning. Start with a copy of E, and filter out
(remove) the edges (u,v) where P[u] = P[v].  (This removes the contracted edges.) Now for
all edges (u,v) that remain, replace (u,v) by (P[u], P[v]), so all of the edges are between
vertices of Vc.

(6) return (Vc, Ec, P)
```

Star Partition Implementation



Algorithm (Star Partitioning): flip a coin for each vertex. If it comes up heads, make it a "star center." If it comes up tails, then connect it to a neighboring star center!

```
fun starPartition(V : sequence<int>, E : sequence<int, int>, n : int) ->
(sequence<int>, sequence<int, int>, sequence<int>):
```

	Work	Span
`heads = tabulate(fn i => random_bool(), n)`	n	1
`P = [0,1,...n-1]`	n	1
`TH = filter (fn (u,v) => heads[u]==false && heads[v]==true, E)`	m	$\log m$
`P = inject (P, TH)` P[u] ← v	$m \log m$	$\log^2 m$
`Vc = filter (fn j => P[j] == j, [0...n-1])`	n	$\log n$
`E' = filter (fn (u,v) => P[u] $\neq$ P[v], E)`	m	$\log m$
`Ec = map (fn (u,v) => (P[u], P[v]), E')`	m	1
`return (Vc, Ec, P)` Total	$(m+n)/\log m$	$\log^2(n+m)$

Star Contraction Cost Analysis

- Putting this all together, we obtain the following bounds on Star Partition:

- Work: $O\big((m+n)\log n\big)$

- Span: $O(\log^2 m)$

Conclusion

- This lecture, we
  - Defined graph contraction and motivated what types of problems it solves
  - Related the contract, recurse, expand structure of Graph Contraction to Contraction Scan
  - Used randomness to outline two graph contraction algorithms:
    - **Edge Contraction**, which generalizes poorly
    - **Star Contraction**, and we also implemented it!

- Next lecture, we will show how to apply star contraction to give an efficient parallel algorithm for computing the contracted components of an undirected graph.