Algorithms at Scale
(Week 8)
Summary

Last Week: Caching

External memory model
• How to predict the performance of algorithms?
B-trees
• Efficient searching
Write-optimized data structures
• Buffer trees
Cache-oblivious algorithms
• van Emde Boas memory layout

Today: Graph Algorithms

Breadth-First-Search
• Sorting your graph
MIS
• Luby’s Algorithm
• Cache-efficient implementation
MST
• Connectivity
• Minimum Spanning Tree
Announcements / Reminders

Today:

MiniProject “proposal” due today.

Next week:

Midterm exam (in class)
Midterm info:

- Will post sample from last year.
- In class, here, 2 hours.
- Material up to (and including) today.
  (Lecture, “tutorial”, problem sets, etc.)
- One double-sided “cheat sheet” allowed

Note:

- I will be out of town.
- Prof. Diptarka Chakraborty will give the exam.
Two types of questions:

1. Algorithms questions
   - For example: sublinear connectivity, streaming distinct elements, B-trees, etc.
   - Know the algorithms... when they are useful... when they are not useful...
   - Understand why they work.

2. Technique questions
   - For example: sampling, reservoir sampling, Chernoff/Hoeffding bounds, median-of-means, etc.
   - Know the techniques, how to use them, when they work (and when they don’t work).
Today’s Problem: Connected Components

Assumptions:

Graph $G = (V,E)$
- Undirected
- $n$ nodes
- $m$ edges
- maximum degree $d$

Error term: $\varepsilon$

Output:
Number of connected components.

Example: output 3
Summary

Last Week: Caching

External memory model
  • How to predict the performance of algorithms?

B-trees
  • Efficient searching

Write-optimized data structures
  • Buffer trees

Cache-oblivious algorithms
  • van Emde Boas memory layout

Today: Graph Algorithms

Breadth-First-Search
  • Sorting your graph

MIS

Luby’s Algorithm

Cache-efficient implementation

MST

Connectivity

Minimum Spanning Tree
Problem: Breadth First Search

Searching a graph:

• undirected graph \( G = (V,E) \)
• source node \( s \)
Problem: Breadth First Search

Searching a graph:

• undirected graph $G = (V,E)$
• source node $s$
• each adjacency list stored as an array (consecutive in memory)

Adjacency List Format:

Example:

$u : a, b, c, v$
$v : a, e, f$
$w : b, c, d, f$
$...$
Problem: Breadth First Search

Searching a graph:

- undirected graph \( G = (V,E) \)
- source node \( s \)

Layer-by-layer...
Algorithm:

- \( L_0 = \{ s \} \)
- Repeat until done: construct \( L_{i+1} \) from \( L_i \)

Key idea: neighbors of \( L_i \) form layer \( L_{i+1} \).

Key idea 2: remove already visited nodes.

\[ L_0 = \{ 1 \} \]
Breadth First Search

Algorithm:

• $L_0 = \{s\}$
• Repeat until done: construct $L_{i+1}$ from $L_i$

This edge cannot exist!

(If it did, node 7 would be in Layer 2.)
Breadth First Search

Algorithm:

• $L_0 = \{s\}$
• Repeat until done:
  construct $L_{i+1}$ from $L_i$

Key idea: neighbors of $L_i$ form layer $L_{i+1}$.

Key idea 2: remove already visited nodes from *only two* layers.

$L_0 = \{1\}$
$L_1 = N(1) = \{2, 3, 4\}$
$L_2 = N(L_1) - L_1 - L_0 = \{5, 6, 7\}$
$L_3 = N(L_2) - L_2 - L_1 = \{7, 8, 9, 10\}$
$L_4 = N(L_3) - L_3 - L_2 = \{11\}$
Breadth First Search

Construct \( L_{i+1} \):

1. \( L_{i+1} = \) neighbors of all nodes in \( L_i \)
2. Sort \( L_{i+1} \).
3. Remove duplicates in \( L_{i+1} \).
4. Scan \( L_i, L_{i+1} \): remove nodes in both.
5. Scan \( L_{i-1}, L_{i+1} \): remove nodes in both.

Invariant: each \( L_i \) is sorted.
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{6, 3, 1, 5, 1, 2, 6, 1, 6\}$

Cost?
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{6, 3, 1, 5, 1, 2, 6, 1, 6\}$

Cost:
$|L_1|/B +$
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{6, 3, 1, 5, 1, 2, 6, 1, 6\}$

Cost:
$|L_1|/B + |L_1| + \ldots$
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{6, 3, 1, 5, 1, 2, 6, 1, 6\}$

Cost:

$|L_1|/B + |L_1|$

$+ \text{edges}(|L_1|)/B$
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{6, 3, 1, 5, 1, 2, 6, 1, 6\}$
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 1, 1, 2, 3, 5, 6, 6, 6\}$
**Breadth First Search**

Example:

\[ L_0 = \{1\} \]
\[ L_1 = \{2, 3, 4\} \]
\[ L_2 = \{1, 1, 1, 2, 3, 5, 6, 6, 6\} \]

Remove duplicates
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 1, 1, 2, 3, 5, 6, 6, 6\}$

$O(\text{edges}(L_1)/B)$
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

$O(\text{edges}(L_1)/B)$
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$  

$L_1 = \{2, 3, 4\}$  

$L_2 = \{1, 2, 3, 5, 6\}$  

Subtract $L_1$. 

Layer 0

Layer 1

Layer 2

Layer 3

Layer 4
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 

Diagram of Breadth First Search with layers 0 to 4.
Example:

$L_0 = \{1\}$  

$L_1 = \{2, 3, 4\}$  

$L_2 = \{1, 2, 3, 5, 6\}$  

Subtract $L_1$. 
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

Subtract $L_1$. 
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 2, 3, 5, 6\}$

$O(\|L_1\|/B + O(edges(L_1)/B))$
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 5, 6\}$

$O(|L_1|/B + O(\text{edges}(L_1)/B))$

Subtract $L_1$. 
Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{1, 5, 6\}$

Subtract $L_0$. 
Example:

L₀ = {1}

L₁ = {2, 3, 4}

L₂ = {1, 5, 6}

Subtract L₀.

O(|L₀|/B + O(edges(L₁)/B))
Breadth First Search

Example:

$L_0 = \{1\}$

$L_1 = \{2, 3, 4\}$

$L_2 = \{5, 6\}$

Subtract $L_0$.

$O(|L_0|/B + O(\text{edges}(L_1)/B))$
Breadth First Search

Cost to construct $L_{i+1}$:

1. $L_{i+1} = \text{neighbors of all nodes in } L_i$
   
   \[ 2|L_i| + \frac{\text{edges}(L_i)}{B} \]

2. Sort $L_{i+1}$.

3. Remove duplicates in $L_{i+1}$.

4. Scan $L_i, L_{i+1}$: remove nodes in both.

5. Scan $L_{i-1}, L_{i+1}$: remove nodes in both.
Breadth First Search

Cost to construct $L_{i+1}$:

1. $L_{i+1} =$ neighbors of all nodes in $L_i$
   
   $2|L_i| + \frac{\text{edges}(L_i)}{B}$

2. Sort $L_{i+1}$.

3. Remove duplicates in $L_{i+1}$.

4. Scan $L_i$, $L_{i+1}$: remove nodes in both.

5. Scan $L_{i-1}$, $L_{i+1}$: remove nodes in both.
Breadth First Search

Cost to construct $L_{i+1}$:

1. $L_{i+1} = \text{neighbors of all nodes in } L_i$
   \[ 2|L_i| + \frac{\text{edges}(L_i)}{B} \]

2. Sort $L_{i+1}$.

3. Remove duplicates in $L_{i+1}$.

4. Scan $L_i, L_{i+1}$: remove nodes in both.

5. Scan $L_{i-1}, L_{i+1}$: remove nodes in both.
**Breadth First Search**

**Cost to construct** $L_{i+1}$:

1. $L_{i+1} = \text{neighbors of all nodes in } L_i$  
   \[ 2|L_i| + \frac{\text{edges}(L_i)}{B} \]

2. Sort $L_{i+1}$.  
   \[ \text{sort}(L_i) \]

3. Remove duplicates in $L_{i+1}$.  
   \[ \frac{\text{edges}(L_i)}{B} \]

4. Scan $L_i, L_{i+1}$: remove nodes in both.  
   \[ \frac{|L_i|}{B} + \frac{\text{edges}(L_i)}{B} \]

5. Scan $L_{i-1}, L_{i+1}$: remove nodes in both.  
   \[ \frac{|L_{i-1}|}{B} + \frac{\text{edges}(L_i)}{B} \]
Breadth First Search

Cost to construct $L_{i+1}$:

1. $L_{i+1} = \text{neighbors of all nodes in } L_i$
   
   $2|L_i| + \frac{\text{edges}(L_i)}{B}$

2. Sort $L_{i+1}$.

3. Remove duplicates in $L_{i+1}$.

4. Scan $L_i, L_{i+1}$: remove nodes in both.

5. Scan $L_{i-1}, L_{i+1}$: remove nodes in both.

Sums to $|V|$ over all levels.
(Every node is in one level.)
Breadth First Search

Cost to construct \( L_{i+1} \):

1. \( L_{i+1} = \) neighbors of all nodes in \( L_i \)
2. Sort \( L_{i+1} \).
3. Remove duplicates in \( L_{i+1} \).
4. Scan \( L_i, L_{i+1} \): remove nodes in both.
5. Scan \( L_{i-1}, L_{i+1} \): remove nodes in both.

\[
2|L_i| + \frac{\text{edges}(L_i)}{B}
\]

Sums to \(|V| \) over all levels. (Every node is in one level.)

\[
\text{sort}(L_i) \quad \text{edges}(L_i)/B
\]

Sums to \(2|E|/B \) over all levels.

\[
\frac{|L_i|}{B} + \frac{\text{edges}(L_i)}{B}
\]

\[
\frac{|L_{i-1}|}{B} + \frac{\text{edges}(L_i)}{B}
\]
Breadth First Search

Cost to construct \( L_{i+1} \):

1. \( L_{i+1} = \text{neighbors of all nodes in } L_i \)
2. Sort \( L_{i+1} \).
3. Remove duplicates in \( L_{i+1} \).
4. Scan \( L_i, L_{i+1} \): remove nodes in both.
5. Scan \( L_{i-1}, L_{i+1} \): remove nodes in both.

\[
\begin{align*}
|L_i|/B + \frac{\text{edges}(L_i)}{B} \\
\frac{2|L_i| + \text{edges}(L_i)}{B} \\
\text{sort}(L_i) \\
\frac{\text{edges}(L_i)}{B} \\
\frac{|L_{i-1}| + \text{edges}(L_i)}{B}
\end{align*}
\]

Sums to \(|V|\) over all levels. (Every node is in one level.)

Sums to \(8|E|/B\) over all levels.
Breadth First Search

Total cost:

\[ O(|V| + |E|/B + \text{sort}(|E|)) \]

1. \( L_{i+1} = \) neighbors of all nodes in \( L_i \)
2. Sort \( L_{i+1} \).
3. Remove duplicates in \( L_{i+1} \).
4. Scan \( L_i, L_{i+1} \): remove nodes in both.
5. Scan \( L_{i-1}, L_{i+1} \): remove nodes in both.

Sums to \(|V|/B\) over all levels.

Sums to \(|E|/B\) over all levels.

Sums to \(|V|\) over all levels.

(Each node is in one level.)

\[ 2|L_i| + \text{edges}(L_i)/B \]
\[ \text{sort}(L_i) \]
\[ |L_i|/B + \text{edges}(L_i)/B \]
\[ |L_{i-1}|/B + \text{edges}(L_i)/B \]
First Search

1. $L_{i+1} = \text{neighbors of all nodes in } L_i$
2. Sort $L_{i+1}$.
3. Remove duplicates in $L_{i+1}$.
4. Scan $L_i, L_{i+1}$: remove nodes in both.
5. Scan $L_{i-1}, L_{i+1}$: remove nodes in both.

Total cost:

$O(|V| + |E|/B + \text{sort}(|E|))$

Sums to $|V|$ over all levels. (Every node is in one level.)

$2|L_i| + \text{edges}(L_i)/B$

Sums to $8|E|/B$ over all levels.

$\text{sort}(E) = O\left(\frac{E}{B} \log_{M/B}(E/B)\right)$

Sums to $2|V|/B$ over all levels.
**Total cost:**

\[ O(|V| + |E|/B + \text{sort}(|E|)) \]

1. \( L_{i+1} = \) neighbors of all nodes in \( L_i \)
2. Sort \( L_{i+1} \).
3. Remove duplicates in \( L_{i+1} \).
4. Scan \( L_i, L_{i+1} \): remove nodes in both.

**Compare to:**

\[ O(|V| + |E|) \]
Problem: Breadth First Search

Can we do better?
Problem: Breadth First Search

Can we do better?

Unlikely in dense graph.
Problem: Breadth First Search

Can we do better?

Unlikely in dense graph.

- If $|E| > B|V|$ and BFS needs to read each edge, then requires at least $|V|$ time.
Problem: Breadth First Search

Can we do better?

Unlikely in dense graph.
- If $|E| > B|V|$ and BFS needs to read each edge, then requires at least $|V|$ time.

Unlikely if adjacency lists are stored separately.
- BFS needs to access each node and each list at least once, so requires $|V|$ time.
Problem: Breadth First Search

Can we do better?

Sparse graph

Store all edges in one array.

\[
O \left( \sqrt{\frac{|V||E|}{B}} + \text{sort}(E) \right)
\]

If \(|E| = O(|V|)\) then:

\[
O \left( \frac{|V|}{B} + \text{sort}(E) \right)
\]
Summary

Today: Graph Algorithms

Breadth-First-Search
- Sorting your graph

MIS
- Luby’s Algorithm
- Cache-efficient implementation

MST
- Connectivity
- Minimum Spanning Tree
Maximal Independent Set

Independent Set:

A set of nodes $S$ so that no two neighbors are in $S$. 
Maximal Independent Set

**Independent Set:**

A set of nodes \( S \) so that no two neighbors are in \( S \).

**Maximal Independent Set:**

An independent set \( S \) where no node can be added.

(Every node has a neighbor in the independent set \( S \).)
Maximal Independent Set

Independent Set:
A set of nodes $S$ so that no two neighbors are in $S$.

Maximal Independent Set:
An independent set $S$ where no node can be added.

*Maximum* Independent Set:
An independent set $S$ of maximum size.
Maximal Independent Set

Independent Set:

A set of nodes $S$ so that no two neighbors are in $S$.

Maximal Independent Set:

An independent set $S$ where no node can be added.

*Maximum* Independent Set:

An independent set $S$ of maximum size.
Maximal Independent Set

Independent Set:

A set of nodes $S$ so that no two neighbors are in $S$.

Maximal Independent Set:

An independent set $S$ where no node can be added.

Maximum Independent Set:

An independent set $S$ of maximum size.

NP-Hard
Maximal Independent Set

Greedy MIS Algorithm:
- $S = \text{empty set}$
- for every node $v$:
  - If no neighbor of $v$ is in $S$, then add $v$ to $S$. 
Maximal Independent Set

Greedy MIS Algorithm:
- $S = \text{empty set}$
- for every node $v$:
  - If no neighbor of $v$ is in $S$, then add $v$ to $S$.

Cost:
$O(|V| + |E|)$
(every access is a cache miss)
Maximal Independent Set

Luby’s Algorithm:
• \( S = \emptyset \)
• Repeat until \( V \) is empty:
  1. Mark each node \( u \) with probability \( 1/2d(u) \).
  2. For each edge \((u,v)\): if both \( u \) and \( v \) are marked:
     - if \( d(u) < d(v) \) then unmark \( u \).
     - else if \( d(v) < d(u) \) then unmark \( v \).
     - else if \( d(u) = d(v) \) then unmark node with smaller id.
  3. Add all marked nodes to \( S \).
  4. Delete from \( V \) every marked node.
  5. Delete from \( V \) every neighbor of marked node.
  6. Delete from \( E \) every edge that no longer exists.
Maximal Independent Set

Luby’s Algorithm:

- \( S = \emptyset \)
- Repeat until \( V \) is empty:
  1. Mark each node \( u \) with probability \( 1/2d(u) \).
  2. For each edge \((u,v)\): if both \( u \) and \( v \) are marked:
     - if \( d(u) < d(v) \) then unmark \( u \).
     - else if \( d(v) < d(u) \) then unmark \( v \).
     - else if \( d(u) = d(v) \) then unmark node with smaller id.
  3. Add all marked nodes to \( S \).
  4. Delete from \( V \) every marked node.
  5. Delete from \( V \) every neighbor of marked node.
  6. Delete from \( E \) every edge that no longer exists.

[Example on the board]
Claim 1:

The set $S$ is a maximal independent set.
Claim 1:
The set $S$ is a maximal independent set.

**independent:**
- only add marked nodes to $S$
- unmark if two neighbors are marked
- delete all neighbors of every node added to $S$
Claim 1:
The set $S$ is a maximal independent set.

maximal:
• only delete a node if added to $S$, or a neighbor is added to $S$
• algorithm terminates when all nodes are deleted $\Rightarrow$ all are in $S$ or have a neighbor in $S$. 
Maximal Independent Set

Luby’s Algorithm:

- \( S = \emptyset \)
- Repeat until \( V \) is empty:
  1. Mark each node \( u \) with probability \( \frac{1}{2}d(u) \).
  2. For each edge \((u,v)\): if both \( u \) and \( v \) are marked:
     - if \( d(u) < d(v) \) then unmark \( u \).
     - else if \( d(v) < d(u) \) then unmark \( v \).
     - else if \( d(u) = d(v) \) then unmark node with smaller id.
  3. Add all marked nodes to \( S \).
  4. Delete from \( V \) every marked node.
  5. Delete from \( V \) every neighbor of marked node.
  6. Delete from \( E \) every edge that no longer exists.
Luby’s Algorithm

Analysis

Define: $E_j =$ edges at start of iteration $j$.

Goal: for some constant $\alpha < 1$, show:

$$E[ E_j \mid E_{j-1}] \leq \alpha E_{j-1}$$

Idea: reduce the number of edges by a constant fraction in each iteration.
Define: node $w$ is good if $\geq 1/3$ neighbors have smaller degree than $w$. 
Define: node $w$ is good if $\geq 1/3$ neighbors have smaller degree than $w$.

Define: edge $(u,v)$ is good if $u$ or $v$ is good.
Luby’s Algorithm

Analysis

Claim: At least half of all edges are good.
Claim: At least half of all edges are good.

Proof:
Orient each edge TO the higher degree node.
Luby’s Algorithm

Claim: At least half of all edges are good.

Proof:
Orient each edge TO the higher degree node.
If v is bad, then: $\frac{2}{3}$ are OUT
$\frac{1}{3}$ are IN

Analysis

good $\Rightarrow$ $\geq \frac{1}{3}$ have smaller degree
Luby’s Algorithm

Claim: At least half of all edges are good.

Proof:
Orient each edge TO the higher degree node.

If v is bad, then:  > 2/3 are OUT
                  ≤ 1/3 are IN

Assign two OUT edges to one IN edge.
(At bad nodes, there are enough OUT...)
**Claim:** At least half of all edges are good.

**Proof:**
Assign two OUT edges to one IN edge.

Each BAD edge \((u,v)\) has \(u\) and \(v\) bad.

Since it is IN to a BAD node, it has 2 edges assigned to it.
Claim: At least half of all edges are good.

Proof:
Assign two OUT edges to one IN edge.

Since it is IN to a BAD node, it has 2 edges assigned to it.

If there are B bad nodes, then $\geq 2B$ edges total in graph.
Claim: At least half of all edges are good.

Proof:
If there are \( B \) bad nodes, then \( \geq 2B \) edges total in graph.

If there are \( > E/2 \) bad nodes, then \( > E \) edges total in graph \( \Rightarrow \) impossible.

\( \Rightarrow > E/2 \) good nodes.
Luby’s Algorithm

Analysis

Claim: If $v$ is good, then:

$$\Pr [\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha$$
Luby’s Algorithm

Claim: If $v$ is good, then:

$$\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha$$

$$\Pr[\text{no nbr of } v \text{ marked}] \leq \Pr[\text{no nbr of } v \text{ with smaller degree marked}]$$

Show at least one neighbor of $v$ with smaller degree is marked!
**Luby’s Algorithm**

**Claim:** If \( v \) is good, then:

\[
\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha
\]

\[
\Pr[\text{no nbr of } v \text{ marked}] \leq \Pr[\text{no nbr of } v \text{ with smaller degree marked}]
\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}]
\]

Nodes are marked independently.
Luby’s Algorithm

Claim: If \( v \) is good, then:

\[
\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha
\]

\[
\Pr[\text{no nbr of } v \text{ marked}] \leq \Pr[\text{no nbr of } v \text{ with smaller degree marked}]
\]

\[
\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}]
\]

\[
\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right)
\]

The probability that a node \( w \) is marked is \( 1/2d(w) \).
Luby’s Algorithm

Claim: If \( v \) is good, then:

\[
\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha
\]

\[
\Pr[\text{no nbr of } v \text{ marked}] \leq \Pr[\text{no nbr of } v \text{ with smaller degree marked}]
\]

\[
\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}]
\]

\[
\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right)
\]

\[
\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(v)}\right)
\]

By assumption, \( d(w) < d(v) \).
Luby’s Algorithm

Claim: If \( v \) is good, then:

\[
\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha
\]

\[
\Pr[\text{no nbr of } v \text{ marked}] \leq \Pr[\text{no nbr of } v \text{ with smaller degree marked}]
\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}]
\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right)
\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(v)}\right)
\leq \left(1 - \frac{1}{2d(v)}\right)^{d(v)/3}
\]

At least \( d(v)/3 \) neighbors with smaller degree because \( v \) is good.
Luby’s Algorithm

Claim: If \( v \) is good, then:

\[
\Pr \text{[nbr of } v \text{ marked]} \geq (1 - e^{-1/6}) = 2\alpha
\]

\[
\Pr \text{[no nbr of } v \text{ marked]} \leq \Pr \text{[no nbr of } v \text{ with smaller degree marked]}
\leq \prod_{\text{w smaller degree nbr of } v} \Pr \text{[w not marked]}
\leq \prod_{\text{w smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right)
\leq \prod_{\text{w smaller degree nbr of } v} \left(1 - \frac{1}{2d(v)}\right)^{d(v)/3}
\leq \left(1 - \frac{1}{2d(v)}\right)^{d(v)/3}
\leq e^{-1/6}
\]

\[(1-1/x)^x \leq e^{-1}\]
Claim: If \( w \) is marked, then:

\[ \Pr \left[ \text{unmark } w \mid w \text{ marked} \right] \leq \frac{1}{2} \]
Luby’s Algorithm

Claim: If \( w \) is marked, then:

\[
\Pr[\text{unmark } w \mid w \text{ marked}] \leq \frac{1}{2}
\]

Only unmark if higher degree neighbor is marked.
Luby’s Algorithm

Claim: If \( w \) is marked, then:

\[
\Pr[\text{unmark } w \mid w \text{ marked}] \leq 1/2
\]

\[
\Pr[\text{unmark } w \mid w \text{ marked}] \leq \Pr[\text{higher degree neighbor of } w \text{ marked}]
\]

\[
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)}
\]

Union bound...
Luby’s Algorithm

Claim: If \( w \) is marked, then:

\[
\Pr \left[ \text{unmark } w \mid w \text{ marked} \right] \leq \frac{1}{2}
\]

\[
\Pr \left[ \text{unmark } w \mid w \text{ marked} \right] \leq \Pr[\text{higher degree neighbor of } w \text{ marked}]
\]

\[
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)}
\]

\[
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)}
\]

By assumption, \( d(w) < d(z) \).
Claim: If \( w \) is marked, then:

\[
\Pr \left[ \text{unmark } w \mid w \text{ marked} \right] \leq \frac{1}{2}
\]

\[
\Pr \left[ \text{unmark } w \mid w \text{ marked} \right] \leq \Pr[\text{higher degree neighbor of } w \text{ marked}]
\]

\[
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)}
\]

\[
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)}
\]

\[
\leq \frac{d(w)}{2d(w)}
\]

Node \( w \) has \( d(w) \) neighbors.
Luby’s Algorithm

Claim: If $w$ is marked, then:

$$\Pr[\text{unmark } w \mid w \text{ marked}] \leq \frac{1}{2}$$

\[
\Pr[\text{unmark } w \mid w \text{ marked}] \leq \Pr[\text{higher degree neighbor of } w \text{ marked}]
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)}
\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)}
\leq \frac{d(w)}{2d(w)} \leq \frac{1}{2}
\]
Luby’s Algorithm

Analysis

**Claim:** If \( v \) is good, then:

\[
\Pr \left[ \text{nbr of } v \text{ marked} \right] \geq (1 - e^{-1/6}) = 2\alpha
\]

**Claim:** If \( w \) is marked, then:

\[
\Pr[\text{stay marked } w \mid \text{marked } w] \geq \frac{1}{2}
\]
Luby’s Algorithm

Analysis

Claim: If $v$ is good, then:
\[
\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha
\]

Claim: If $w$ is marked, then:
\[
\Pr[\text{stay marked } w \mid \text{marked } w] \geq \frac{1}{2}
\]

Claim: If $v$ is good, then:
\[
\Pr[\text{node } w, \text{nbr of } v, \text{enters the MIS}] \geq \alpha
\]
Luby’s Algorithm

Analysis

Claim: If $v$ is good, then:
$$\Pr[\text{nbr of } v \text{ marked}] \geq (1 - e^{-1/6}) = 2\alpha$$

Claim: If $w$ is marked, then:
$$\Pr[\text{stay marked } w \mid \text{marked } w] \geq \frac{1}{2}$$

Claim: If $v$ is good, then:
$$\Pr[v \text{ is deleted at end of iteration}] \geq \alpha$$
Claim: If $v$ is good, then:

$$\Pr[v \text{ is deleted at end of iteration}] \geq \alpha$$

Claim: If edge $(u,v)$ is good, then:

$$\Pr[(u,v) \text{ is deleted at end of iteration}] \geq \alpha$$

Because either $u$ or $v$ is good.
Luby’s Algorithm

---

**Analysis**

--

**Claim:** If \( v \) is good, then:

\[
\text{Pr}[v \text{ is deleted at end of iteration}] \geq \alpha
\]

--

**Claim:** If edge \((u,v)\) is good, then:

\[
\text{Pr}[(u,v) \text{ is deleted at end of iteration}] \geq \alpha
\]

--

\[
\mathbb{E}[E_j | E_{j-1}] \leq E_{j-1}(1 - \alpha/2)
\]
Luby’s Algorithm

Analysis

\[ \mathbb{E}[E_j | E_{j-1}] \leq E_{j-1} (1 - \alpha/2) \]

\[ \mathbb{E}[E_j] = \mathbb{E}[\mathbb{E}[E_j | E_{j-1}]] \]

Law of Total Expectation
Luby’s Algorithm

Analysis

\[ E[E_j | E_{j-1}] \leq E_{j-1}(1 - \alpha/2) \]

\[ E[E_j] = E[E[E_j | E_{j-1}]] \leq E[E_{j-1}](1 - \alpha/2) \]

Substitution.
Luby’s Algorithm

Analysis

\[ \mathbb{E}[E_j | E_{j-1}] \leq E_{j-1}(1 - \alpha/2) \]

\[ \mathbb{E}[E_j] = \mathbb{E}[\mathbb{E}[E_j | E_{j-1}]] \]
\[ \leq \mathbb{E}[E_{j-1}](1 - \alpha/2) \]
\[ \leq |E|(1 - \alpha/2)^j \]

Induction.
Note that \( E_0 = |E| \).
Luby’s Algorithm

Analysis

\[ E[E_j|E_{j-1}] \leq E_{j-1}(1 - \alpha/2) \]

\[ E[E_j] = E[E[E_j|E_{j-1}]] \leq E[E_{j-1}](1 - \alpha/2) \leq |E|(1 - \alpha/2)^j \]

\[ E[\text{iterations}] \leq O \left( \frac{2}{\alpha} \log(|E|) \right) \]

Prove this. (Hint: Markov’s Inequality is useful.)
Theorem:

Luby’s Algorithm terminates in $O(\log |E|)$ iterations, in expectation.
Luby’s Algorithm

Expected time?
Maximal Independent Set

Luby’s Algorithm:

• $S = \emptyset$

• Repeat until $V$ is empty:
  1. Mark each node $u$ with probability $\frac{1}{2d(u)}$.
  2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
     - if $d(u) < d(v)$ then unmark $u$.
     - else if $d(v) < d(u)$ then unmark $v$.
     - else if $d(u) = d(v)$ then unmark node with smaller id.
  3. Add all marked nodes to $S$.
  4. Delete from $V$ every marked node.
  5. Delete from $V$ every neighbor of marked node.
  6. Delete from $E$ every edge that no longer exists.
Luby’s Algorithm

Expected time?

\[ O(E + (1 - \alpha/2)E + (1 - \alpha/2)^2 E + (1 - \alpha/2)^3 E + \ldots) = O(E) \]
Theorem:
Luby’s Algorithm terminates in $O(\log |E|)$ iterations, in $O(E)$ time, in expectation.
Luby’s Algorithm:

- \( S = \emptyset \)
- Repeat until \( V \) is empty:
  1. Mark each node \( u \) with probability \( \frac{1}{2d(u)} \).
  2. For each edge \((u,v)\): if both \( u \) and \( v \) are marked:
      - if \( d(u) < d(v) \) then unmark \( u \).
      - else if \( d(v) < d(u) \) then unmark \( v \).
      - else if \( d(u) = d(v) \) then unmark node with smaller id.
  3. Add all marked nodes to \( S \).
  4. Delete from \( V \) every marked node.
  5. Delete from \( V \) every neighbor of marked node.
  6. Delete from \( E \) every edge that no longer exists.
Cache-Efficient Luby’s

Setup

Initially:

Assume that all the edges are in a single array.

Ex:

$$[(u,v), (u,w), (x,z), (z,u), (x,w)]$$

This could take $O(|V|)$ time to construct, otherwise.
Initially:

Assume that all the edges are in a single array. Assume each edge also stores:

- \( \text{deg}(u), \text{deg}(v) \)
- 1-bit: marked
- 1-bit: deleted

Ex:

\[
[(u,v,3,3,00), (u,w,2,4,00), (x,z,4,2,00), (z,u,5,2,00), (x,w,3,1,00)]
\]
Cache-Efficient Luby’s

Setup

Initially:

Assume that all the edges are in a single array.
Assume each edge also stores:
• $\deg(u)$, $\deg(v)$
• 1-bit: marked
• 1-bit: deleted
Assume each edge is stored twice: $(u,v)$ and $(v,u)$

Ex:
$[(u,v),(v,u),(u,w),(w,u),(x,z),(z,x),(z,u),(u,z)]$
Initially:

Assume that all the edges are in a single array. Assume each edge also stores:
- $\deg(u)$, $\deg(v)$
- 1-bit: marked
- 1-bit: deleted

Assume each edge is stored twice: $(u,v)$ and $(v,u)$

To access the edges adjacent to $u$: sort the edge array.
Cache Efficient Luby’s

Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.
2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
   - if $d(u) < d(v)$ then unmark $u$.
   - else if $d(v) < d(u)$ then unmark $v$.
   - else if $d(u) = d(v)$ then unmark node with smaller id.
3. Add all marked nodes to $S$.
4. Delete from $V$ every marked node.
5. Delete from $V$ every neighbor of marked node.
6. Delete from $E$ every edge that no longer exists.
Cache Efficient Luby’s

Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.

Cache-efficient:
Sort the array by node.
Scan the array.
For each node $u$, flip a random coin to decide on mark.
(Use the degree of each node that is stored with the edge.)
Set the mark bits for each edge $(u, .)$.

$O(sort(E) + E/B)$
Cache Efficient Luby’s

Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.
2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
   - if $d(u) < d(v)$ then unmark $u$.
   - else if $d(v) < d(u)$ then unmark $v$.
   - else if $d(u) = d(v)$ then unmark node with smaller id.

Cache-efficient:
Make a copy $E'$.
Sort by $2^{nd}$ component of edge $(., u)$.
Iterate and unmark if higher degree neighbor is marked.
Cache Efficient Luby’s

Sort by first:

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Sort by second:

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Cache Efficient Luby’s

**Sort by first:**

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>2</td>
<td>1</td>
<td>X</td>
<td>2</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Sort by second:**

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Scan neighbors of node a.
Do not unmark a.
Scan neighbors of node b.
If b were marked, unmark b because a is marked.
Cache Efficient Luby’s

Sort by first:

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td><strong>1</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort by second:

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan neighbors of node c.
None are marked.
Cache Efficient Luby’s

Sort by first:

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Sort by second:

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan neighbors of node d.
Cache Efficient Luby’s

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort by first:

Sort by second:

Scan neighbors of node e.
Unmark e because a is marked and has higher degree.
Cache Efficient Luby’s

<table>
<thead>
<tr>
<th></th>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>second</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Sort by second:

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>second</td>
<td>2</td>
<td>X</td>
<td>X</td>
<td>3</td>
<td>X</td>
<td>X</td>
<td>2</td>
<td>X</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Scan neighbors of node e.  
Unmark e because a is marked and has higher degree.
Cache Efficient Luby’s

Sort by first:

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,d)</th>
<th>(a,e)</th>
<th>(b,a)</th>
<th>(b,c)</th>
<th>(c,b)</th>
<th>(d,a)</th>
<th>(d,e)</th>
<th>(e,a)</th>
<th>(e,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort by second:

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(d,a)</th>
<th>(e,a)</th>
<th>(a,b)</th>
<th>(c,b)</th>
<th>(b,c)</th>
<th>(a,d)</th>
<th>(e,d)</th>
<th>(a,e)</th>
<th>(d,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ O(\text{sort}(E) + E/B) \]
Cache Efficient Luby’s

Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.
2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
   - if $d(u) < d(v)$ then unmark $u$.
   - else if $d(v) < d(u)$ then unmark $v$.
   - else if $d(u) = d(v)$ then unmark node with smaller id.

Cache-efficient:
Make a copy $E'$.
Sort by $2^{nd}$ component of edge $(., u)$.
Iterate and unmark if higher degree neighbor is marked.
Luby’s Iteration:
1. Mark each node \( u \) with probability \( 1/2d(u) \).
2. For each edge \((u,v)\): if both \( u \) and \( v \) are marked:
   - if \( d(u) < d(v) \) then unmark \( u \).
   - else if \( d(v) < d(u) \) then unmark \( v \).
   - else if \( d(u) = d(v) \) then unmark node with smaller id.
3. Add all marked nodes to \( S \).
4. Delete from \( V \) every marked node.

Cache-efficient:
Create two new arrays \( S \) and (new) \( E \).
Copy all marked edges into \( S \) and all unmarked edges into (new) \( E \).

\( O(E/B) \)
Cache Efficient Luby’s

Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.
2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
   - if $d(u) < d(v)$ then unmark $u$.
   - else if $d(v) < d(u)$ then unmark $v$.
   - else if $d(u) = d(v)$ then unmark node with smaller id.
3. Add all marked nodes to $S$.
4. Delete from $V$ every marked node.
5. Delete from $V$ every neighbor of marked node.
6. Delete from $E$ every edge that no longer exists.

Cache-efficient:
Sort $S$. Sort $E$.
Scan and delete from $E$. 
Cache Efficient Luby’s

E (sorted by second)

\[
\begin{array}{cccccccc}
(b,a) & (c,a) & (e,a) & (b,d) & (h,d) & (d,f) & (c,f) & (d,h) \\
2 & 2 & 2 & 1 & 2 & 1 & 2 & \\
\end{array}
\]

S (sorted by first)

\[
\begin{array}{cccccccc}
(a,b) & (a,c) & (a,e) & (f,d) & (f,c) & \\
3 & 3 & 3 & 2 & 2 & \\
x & x & x & x & x & \\
\end{array}
\]

Scan neighbors of node a.
Mark to delete if neighbor is marked.
### Cache Efficient Luby’s

#### E (sorted by second)

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Type</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### S (sorted by first)

<table>
<thead>
<tr>
<th></th>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,e)</th>
<th>(f,d)</th>
<th>(f,c)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan neighbors of node a.
Mark to delete if neighbor is marked.
Cache Efficient Luby’s

E (sorted by second)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,e)</th>
<th>(f,d)</th>
<th>(f,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Scan neighbors of node d.
Mark to delete if neighbor is marked.
Cache Efficient Luby’s

E (sorted by second)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,e)</th>
<th>(f,d)</th>
<th>(f,c)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan neighbors of node f.
Mark to delete if neighbor is marked.
Cache Efficient Luby’s

<table>
<thead>
<tr>
<th>E (sorted by second)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b,a)</td>
<td>2</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c,a)</td>
<td>2</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e,a)</td>
<td>2</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b,d)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h,d)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d,f)</td>
<td>2</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c,f)</td>
<td>1</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d,h)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S (sorted by first)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b)</td>
<td>3</td>
<td></td>
<td></td>
<td>(f,d)</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a,c)</td>
<td>3</td>
<td></td>
<td></td>
<td>(f,c)</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a,e)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f,d)</td>
<td>2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f,c)</td>
<td>2</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan neighbors of node f.  
Mark to delete if neighbor is marked.
Cache Efficient Luby’s

E (sorted by second)

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

S (sorted by first)

<table>
<thead>
<tr>
<th></th>
<th>(a,b)</th>
<th>(a,c)</th>
<th>(a,e)</th>
<th>(f,d)</th>
<th>(f,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Scan neighbors of node h.
Mark to delete if neighbor is marked.
Cache Efficient Luby’s

E (sorted by second)

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Sort and mark all associated with same node as deleted.
Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

E (sorted by second)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Copy and sort.
Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E (sorted by second)

<table>
<thead>
<tr>
<th></th>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan and mark deleted if any neighbor is marked deleted.
Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

E (sorted by second)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Scan and mark deleted if any neighbor is marked deleted.
**Cache Efficient Luby’s**

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

E (sorted by second)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(c,a)</th>
<th>(e,a)</th>
<th>(b,d)</th>
<th>(h,d)</th>
<th>(d,f)</th>
<th>(c,f)</th>
<th>(d,h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Scan and mark deleted if any neighbor is marked deleted.
Cache Efficient Luby’s

E (sorted by first)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b,a)</td>
<td>(b,d)</td>
<td>(c,a)</td>
<td>(c,f)</td>
<td>(d,f)</td>
<td>(d,h)</td>
<td>(e,a)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

E (sorted by second)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b,a)</td>
<td>(c,a)</td>
<td>(e,a)</td>
<td>(b,d)</td>
<td>(h,d)</td>
<td>(d,f)</td>
<td>(c,f)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>1</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Scan and mark deleted if any neighbor is marked deleted.
Cache Efficient Luby’s

E (sorted by first)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b,a)</td>
<td>(b,d)</td>
<td>(c,a)</td>
<td>(c,f)</td>
<td>(d,f)</td>
<td>(d,h)</td>
<td>(e,a)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

E (sorted by second)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(b,a)</td>
<td>(c,a)</td>
<td>(e,a)</td>
<td>(b,d)</td>
<td>(h,d)</td>
<td>(d,f)</td>
<td>(c,f)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Scan and mark deleted if any neighbor is marked deleted.
Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

new array

Copy anything left to a new array E for the next iteration.
### Cache Efficient Luby’s

E (sorted by first)

<table>
<thead>
<tr>
<th>(b,a)</th>
<th>(b,d)</th>
<th>(c,a)</th>
<th>(c,f)</th>
<th>(d,f)</th>
<th>(d,h)</th>
<th>(e,a)</th>
<th>(h,d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

new array

\[ O(\text{sort}(E) + E/B) \]
Luby’s Iteration:
1. Mark each node $u$ with probability $1/2d(u)$.
2. For each edge $(u,v)$: if both $u$ and $v$ are marked:
   - if $d(u) < d(v)$ then unmark $u$.
   - else if $d(v) < d(u)$ then unmark $v$.
   - else if $d(u) = d(v)$ then unmark node with smaller id.
3. Add all marked nodes to $S$.
4. Delete from $V$ every marked node.
5. Delete from $V$ every neighbor of marked node.
6. Delete from $E$ every edge that no longer exists.

Cache-efficient:

$$O(sort(E) + E/B)$$
Luby’s Algorithm

Analysis

Theorem:

Luby’s Algorithm terminates in $O(\log |E|)$ iterations, in $O(E/B + \text{sort}(E))$ time, in expectation.

\[
\text{sort}(E) = O\left(\frac{E}{B} \log_{M/B}(E/B)\right)
\]
Summary

Today: Graph Algorithms

Breadth-First-Search
•  *Sorting your graph*

MIS
•  *Luby’s Algorithm*
•  *Cache-efficient implementation*

MST
•  *Connectivity*
•  *Minimum Spanning Tree*
Connected Components

Idea: Transform graph into depth-1 trees.
Cache-Efficient Connectivity

Setup

Initially:

Assume that all the edges are in a single array.
Assume each edge is stored ONCE

Ex:

\[\{(u,v),(u,w),(x,z),(z,u)\}\]
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.

Base case:
One edge $\Rightarrow$ done.
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow \text{depth 1 trees}$.

Only "root" nodes in $E_2$ are connected to $E_1$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.

Claim: does not change connected components.
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.

Claim: does not change connected components.
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.

Claim: does not change connected components.
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.

![Diagram showing the division and contraction process]
Cache-Efficient Connectivity

Algorithm Idea

1. Divide \( E \) into two parts: \( E_1 \) and \( E_2 \).
2. Recursively solve \( E_2 \rightarrow \) depth 1 trees.
3. Contract \( E_1 \).

Claim: does not change connected components.

Algorithm:
For each \((x,y)\) in \( E_1 \): if \((a,x)\) or \((a,y)\) is in \( E_2 \) then:
Replace \((x,y)\) with \((y,a)\) or \((x,y)\) with \((x,a)\).
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.

Only “root” nodes in $E_2$ are connected to $E_1$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$.

No merging necessary!
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 

Cache-Efficient Connectivity

Algorithm Idea

E2 depth-1 tree
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 

Cache-Efficient Connectivity

Algorithm Idea

---

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 

Cache-Efficient Connectivity

Algorithm Idea

---

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 

Cache-Efficient Connectivity

Algorithm Idea

---
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$.

Claim: Does not change connected components.

**Algorithm:**

For each $(a,b)$ in $E_2$:
- If $a$ is an $E_1$ root: add $(a,b)$ to $E_1$.
- Else if $(x,a)$ in $E_1$: add $(x,b)$ to $E_1$. 
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 

Diagram showing the divide and conquer process.
Cache-Efficient Connectivity

Contract(E1, E2)

1. Sort E1 by first.
2. Sort E2 by second.
3. Scan: (a,b) in E1, (x,a) in E2 $\Rightarrow$ delete(a,b), add(x,b)

4. Sort E1 by second.
5. Sort E2 by second.
6. Scan: (a,b) in E1, (x,b) in E2 $\Rightarrow$ delete(a,b), add(x,a)
Cache Efficient Contract

<table>
<thead>
<tr>
<th>E1 (sorted by first)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b)</td>
<td>(b,d)</td>
<td>(b,c)</td>
<td>(c,e)</td>
<td>(c,f)</td>
<td>(d,g)</td>
<td>(d,h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E2 (sorted by second)</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(z,b)</td>
<td>(z,c)</td>
<td>(y,d)</td>
<td>(y,f)</td>
<td>(z,j)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sort E1 by first, E2 by second.
Cache Efficient Contract

E1 (sorted by first)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b)</td>
<td>(b,d)</td>
<td>(b,c)</td>
<td>(c,e)</td>
<td>(c,f)</td>
<td>(d,g)</td>
<td>(d,h)</td>
</tr>
</tbody>
</table>

E2 (sorted by second)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(z,b)</td>
<td>(z,c)</td>
<td>(y,d)</td>
<td>(y,f)</td>
<td>(z,j)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scan: look for (b, .)
Cache Efficient Contract

### E1 (sorted by first)

| (a,b)  | (b,d)  | (b,c)  | (c,e)  | (c,f)  | (d,g)  | (d,h)  |

### E2 (sorted by second)

| (z,b)  | (z,c)  | (y,d)  | (y,f)  | (z,j)  |        |        |

Scan: look for (b, .)
### Cache Efficient Contract

**E1 (sorted by first)**

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(b,c)</th>
<th>(c,e)</th>
<th>(c,f)</th>
<th>(d,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

**E2 (sorted by second)**

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

**Scan:** replace \((b,d)\) with \((z,d)\)
Cache Efficient Contract

E1 (sorted by first)

| (a,b) | (z,d) | (b,c) | (c,e) | (c,f) | (d,g) | (d,h) |

E2 (sorted by second)

| (z,b) | (z,c) | (y,d) | (y,f) | (z,j) |

Scan: replace (b,c) with (z,c)
Cache Efficient Contract

E1 (sorted by first)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b)</td>
<td>(z,d)</td>
<td>(z,c)</td>
<td>(c,e)</td>
<td>(c,f)</td>
<td>(d,g)</td>
</tr>
</tbody>
</table>

E2 (sorted by second)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(z,b)</td>
<td>(z,c)</td>
<td>(y,d)</td>
<td>(y,f)</td>
<td>(z,j)</td>
</tr>
</tbody>
</table>

Scan: replace (b,c) with (z,c)
### Cache Efficient Contract

**E1 (sorted by first)**

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(c,e)</th>
<th>(c,f)</th>
<th>(d,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

**E2 (sorted by second)**

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

**Scan...**
Cache Efficient Contract

E1 (sorted by first)

| (a,b) | (z,d) | (z,c) | (z,e) | (c,f) | (d,g) | (d,h) |

E2 (sorted by second)

| (z,b) | (z,c) | (y,d) | (y,f) | (z,j) |
### Cache Efficient Contract

**E1 (sorted by first)**

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(z,e)</th>
<th>(c,f)</th>
<th>(d,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

**E2 (sorted by second)**

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Scan...
Cache Efficient Contract

E1 (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(z,e)</th>
<th>(z,f)</th>
<th>(d,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

E2 (sorted by second)

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

Replace...
Cache Efficient Contract

E1 (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(z,e)</th>
<th>(z,f)</th>
<th>(d,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

E2 (sorted by second)

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

Scan...
Cache Efficient Contract

E1 (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(z,e)</th>
<th>(z,f)</th>
<th>(y,g)</th>
<th>(d,h)</th>
</tr>
</thead>
</table>

E2 (sorted by second)

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

Replace...
Cache Efficient Contract

E1 (sorted by first)

| (a,b) | (z,d) | (z,c) | (z,e) | (z,f) | (y,g) | (d,h) |

E2 (sorted by second)

| (z,b) | (z,c) | (y,d) | (y,f) | (z,j) |

Scan...
Cache Efficient Contract

E1 (sorted by first)

<table>
<thead>
<tr>
<th>(a,b)</th>
<th>(z,d)</th>
<th>(z,c)</th>
<th>(z,e)</th>
<th>(z,f)</th>
<th>(y,g)</th>
<th>(y,h)</th>
</tr>
</thead>
</table>

E2 (sorted by second)

<table>
<thead>
<tr>
<th>(z,b)</th>
<th>(z,c)</th>
<th>(y,d)</th>
<th>(y,f)</th>
<th>(z,j)</th>
</tr>
</thead>
</table>

Replace...
Cache-Efficient Connectivity

Contract(E1, E2)

1. Sort E1 by first.
2. Sort E2 by second.
3. Scan: (a,b) in E1, (x,a) in E2 \implies delete(a,b), add(x,b)

4. Sort E1 by second.
5. Sort E2 by second.
6. Scan: (a,b) in E1, (x,b) in E2 \implies delete(a,b), add(x,a)

\( O(sort(E) + E/B) \)
Cache-Efficient Connectivity

---

**Merge(E1, E2)**

1. Sort E1 by second.
2. Sort E2 by first.
3. Scan: (a,b) in E1, (b,c) in E2 \(\Rightarrow\) add(a,c) to E1

4. Sort E1 by first.
5. Sort E2 by first.
6. Scan: (a,.) in E1, (a,x) in E2 \(\Rightarrow\) add(a,x) to E1

\[O(sort(E) + E/B)\]
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$. 
1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2$ \( \Rightarrow \) depth 1 trees.
4. Recursively solve $E_1$ \( \Rightarrow \) depth 1 trees.
5. Merge $E_2$ into $E_1$. 

$O(sort(E) + E/B)$
Cache-Efficient Connectivity

Algorithm Idea

1. Divide $E$ into two parts: $E_1$ and $E_2$.
2. Recursively solve $E_2 \Rightarrow$ depth 1 trees.
4. Recursively solve $E_1 \Rightarrow$ depth 1 trees.
5. Merge $E_2$ into $E_1$.

$$T(E) = 2T(E/2) + O(E/B) + \text{sort}(E)$$
$$= O(\text{sort}(E) \log(E))$$

Faster than BFS (except in sparse case)!
Today: Graph Algorithms

Breadth-First-Search
  • *Sorting your graph*
MIS
  • *Luby’s Algorithm*
  • *Cache-efficient implementation*
MST
  • *Connectivity*
  • *Minimum Spanning Tree*
1. Let $e$ be a random edge.
2. Divide $E$ into two parts:
   - $E_1$ has edges with weight $< w(e)$.
   - $E_2$ has edges with weight $> w(e)$
3. Recursively find MST of $E_1$.
4. Do something.
5. Recursively find MST of $E_2$.
6. Do something.