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)

Announcements / Reminders

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.

Midterm Advice

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:

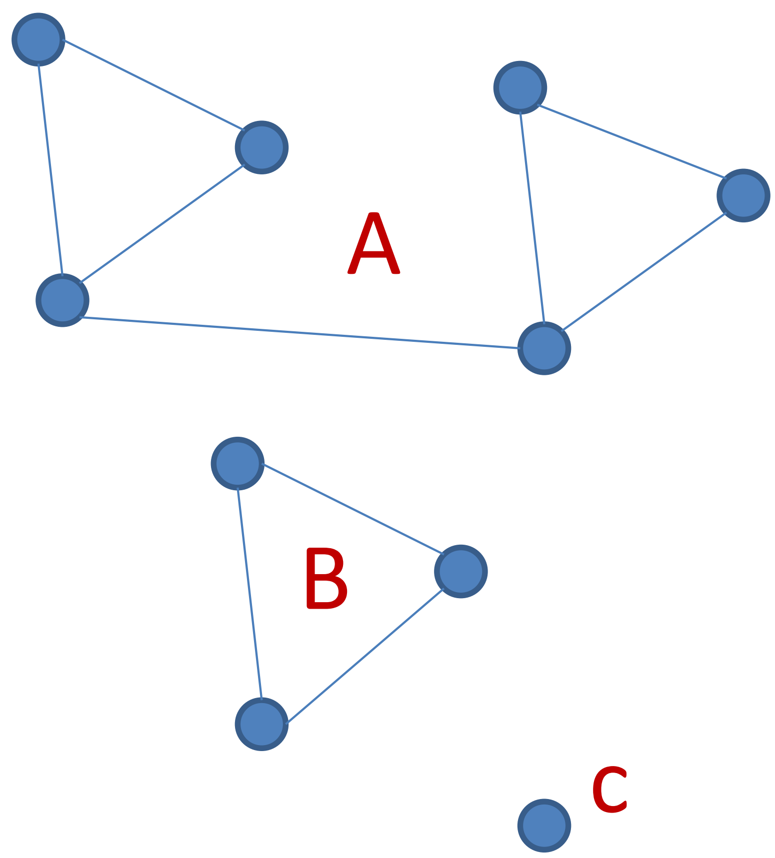
Graph G = (V,E)

- Undirected
- n nodes
- m edges
- maximum degree d

Error term: ε

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*

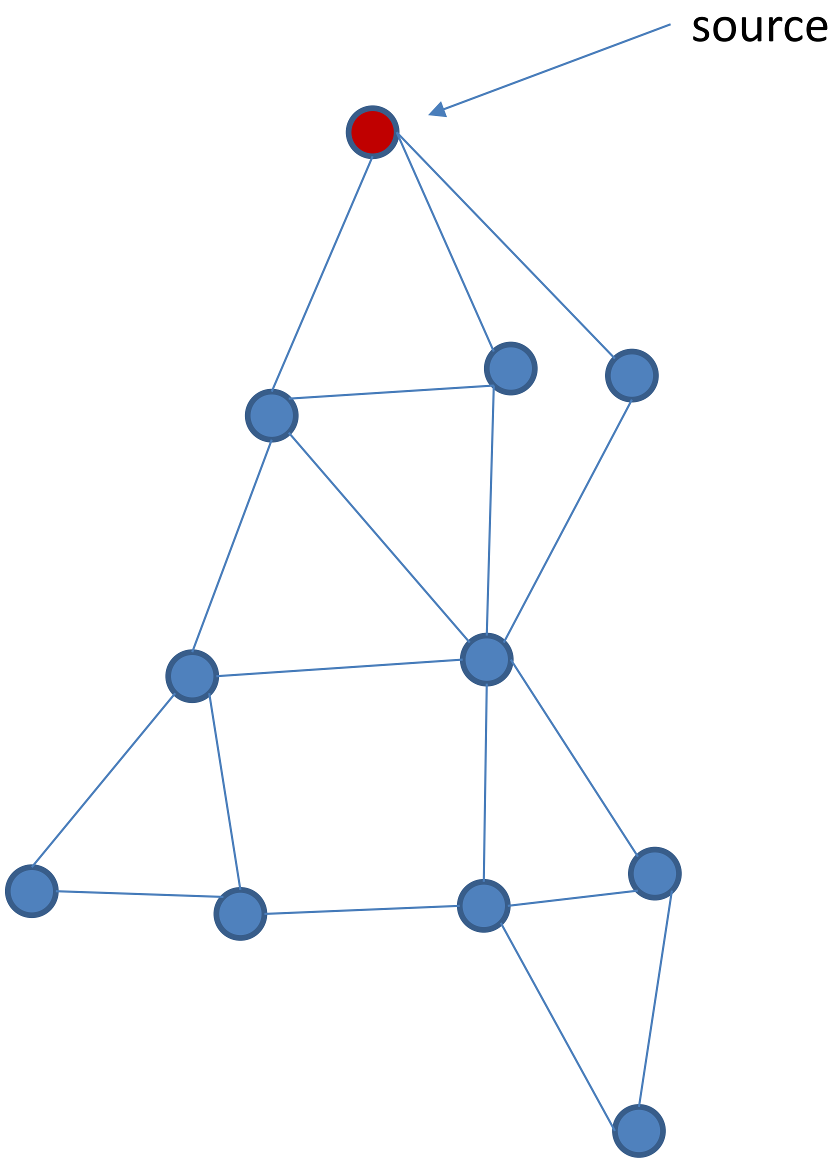
MST

- *Connectivity*
- *Minimum Spanning Tree*

Problem: Breadth First Search

Searching a graph:

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



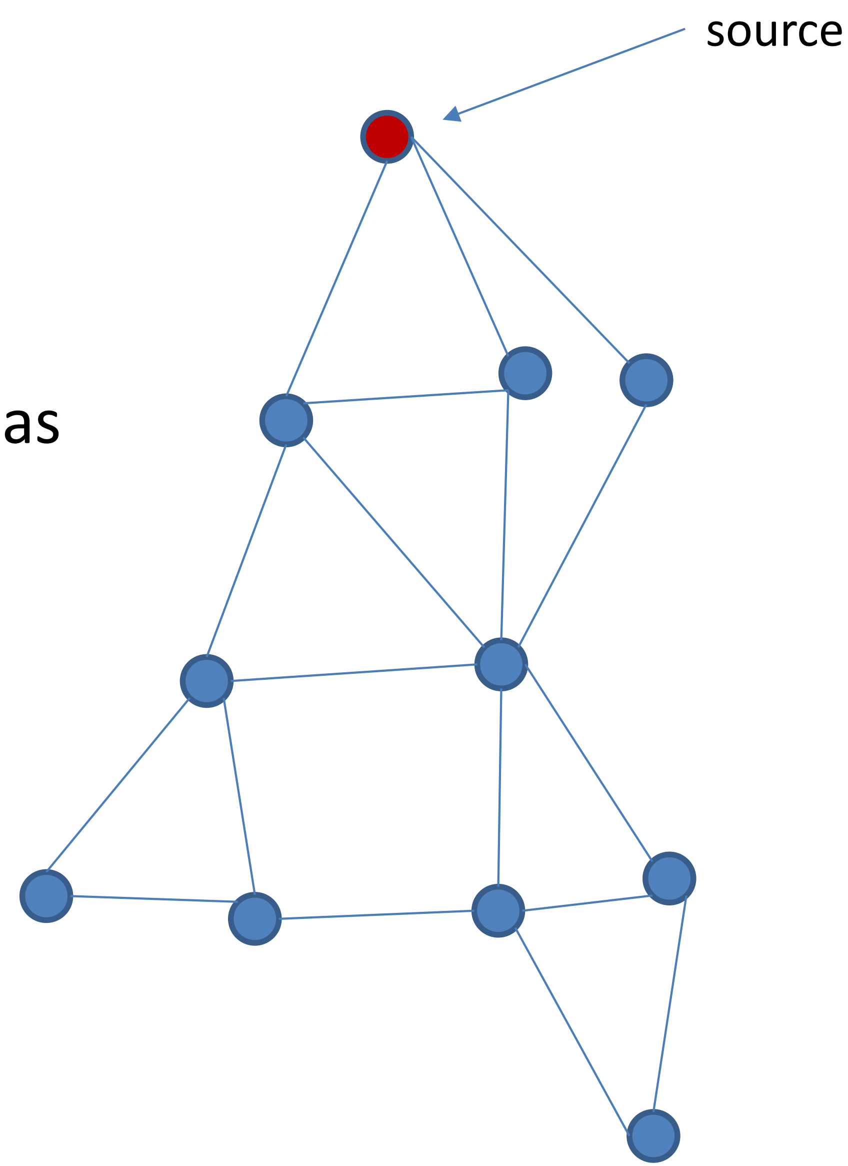
Problem: Breadth First Search

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
…

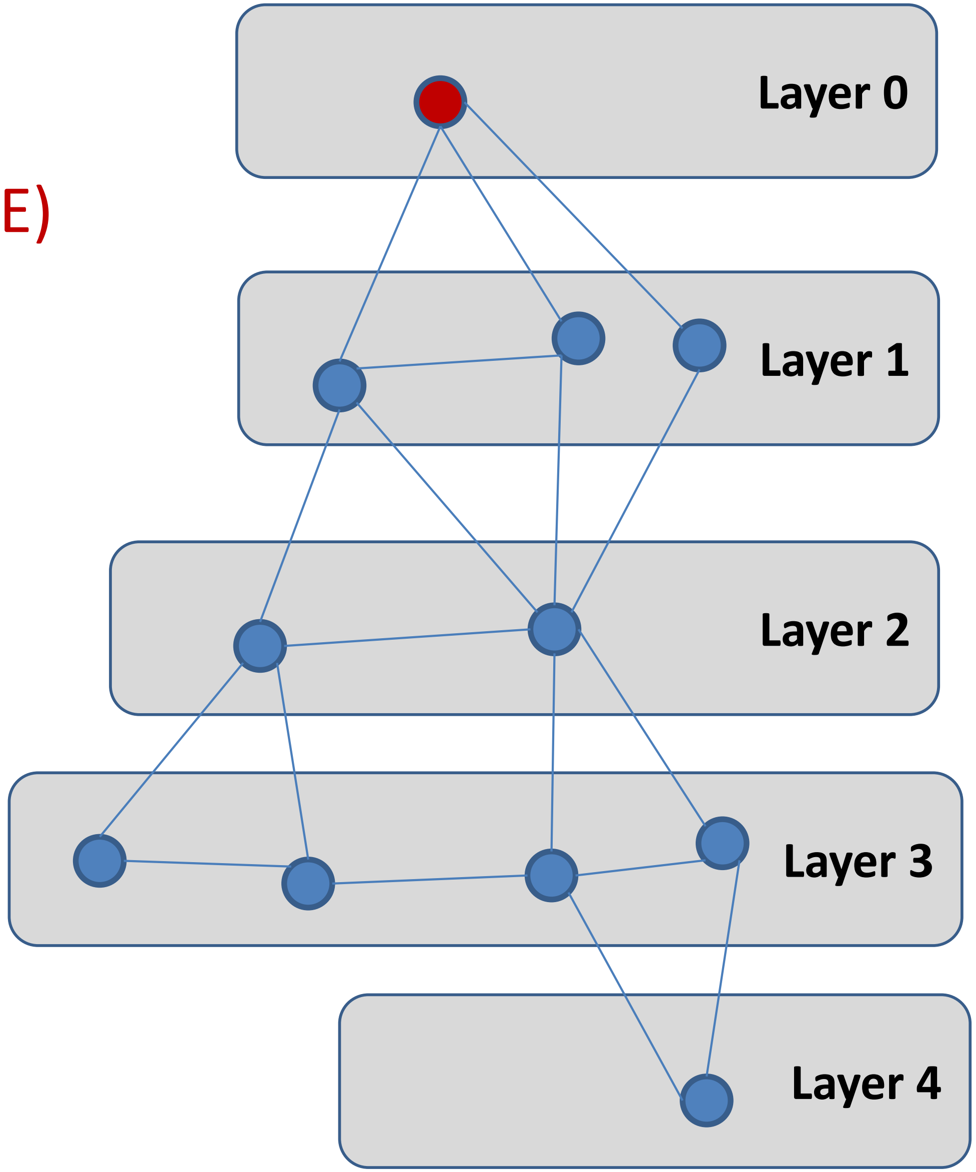


Problem: Breadth First Search

Searching a graph:

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

Layer-by-layer…



Breadth First Search

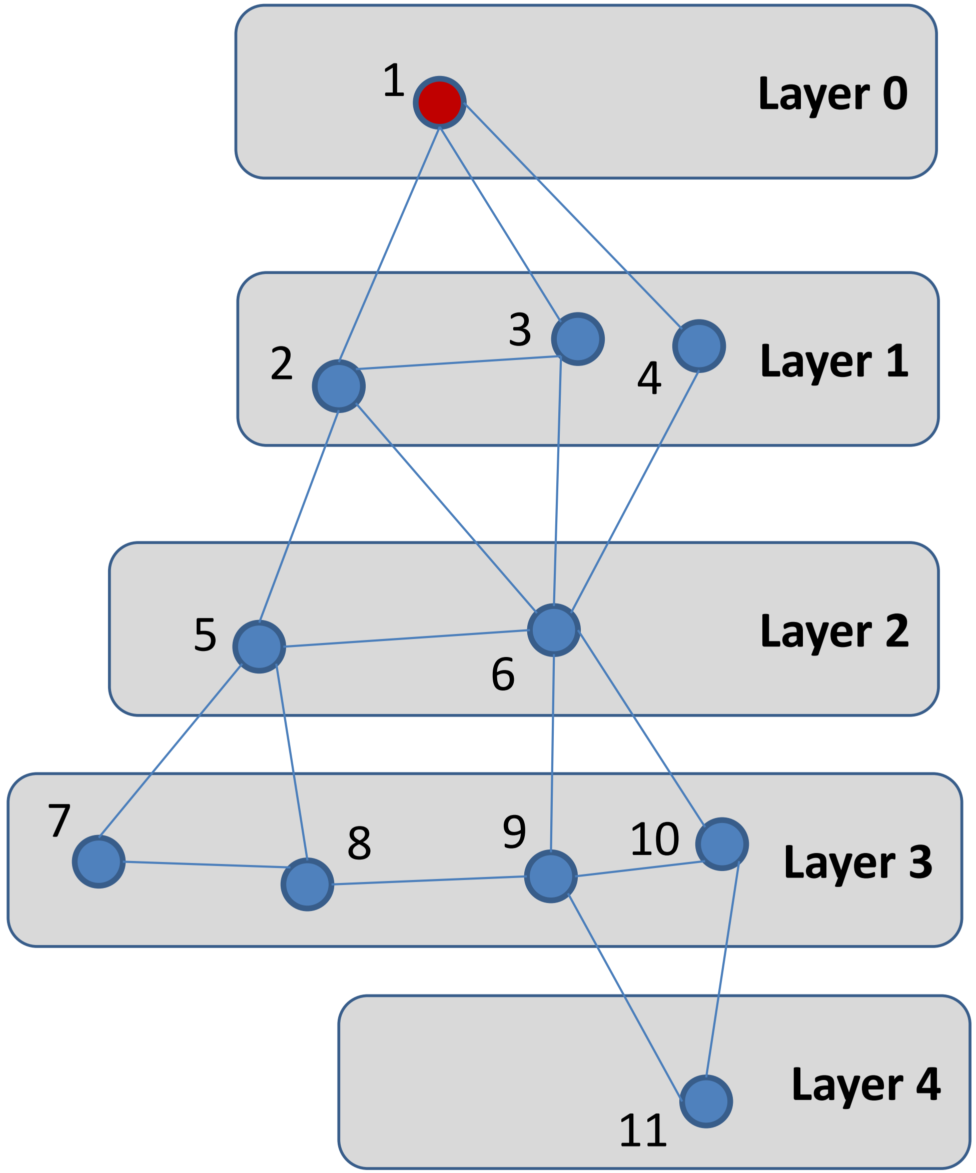
Algorithm:

- 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.

$$L_0 = \{1\}$$



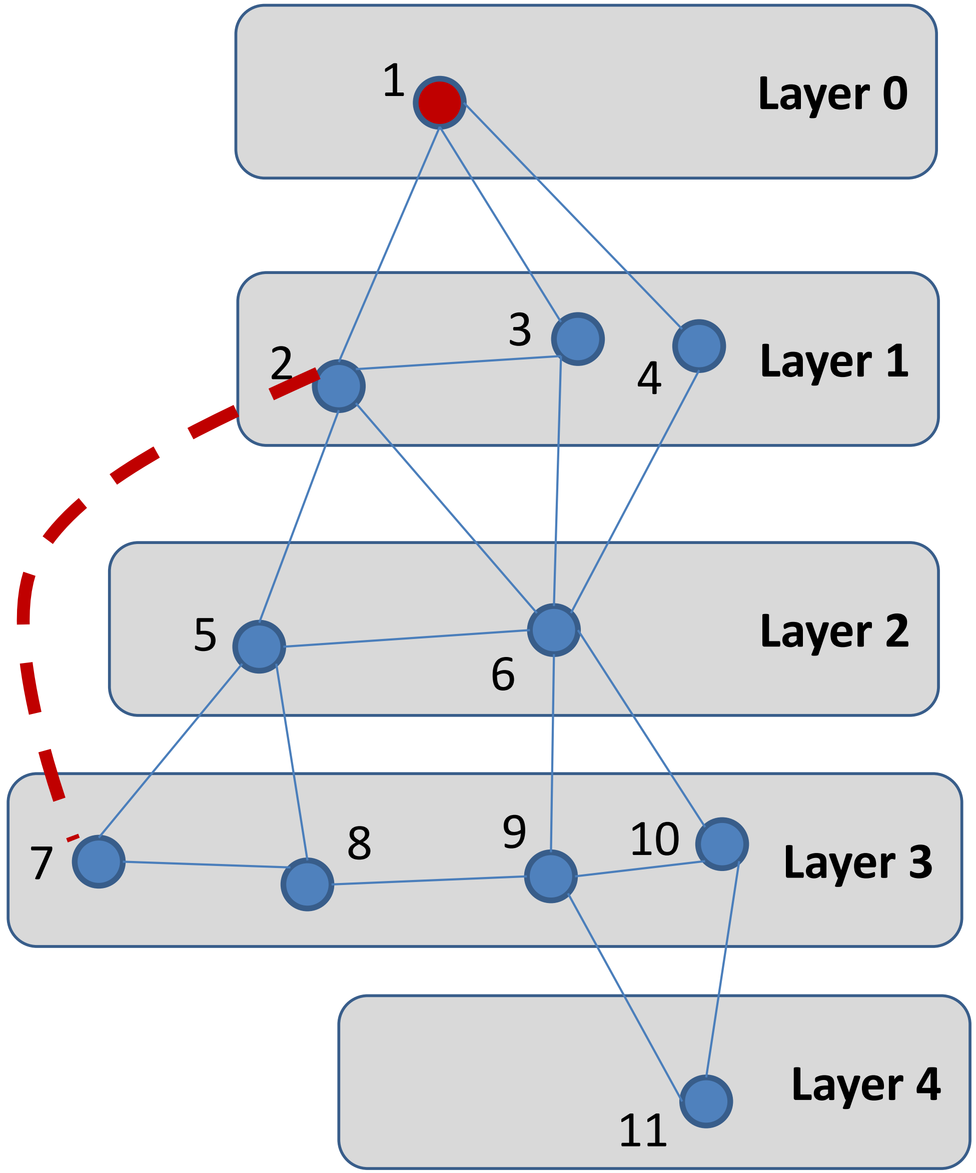
Breadth First Search

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

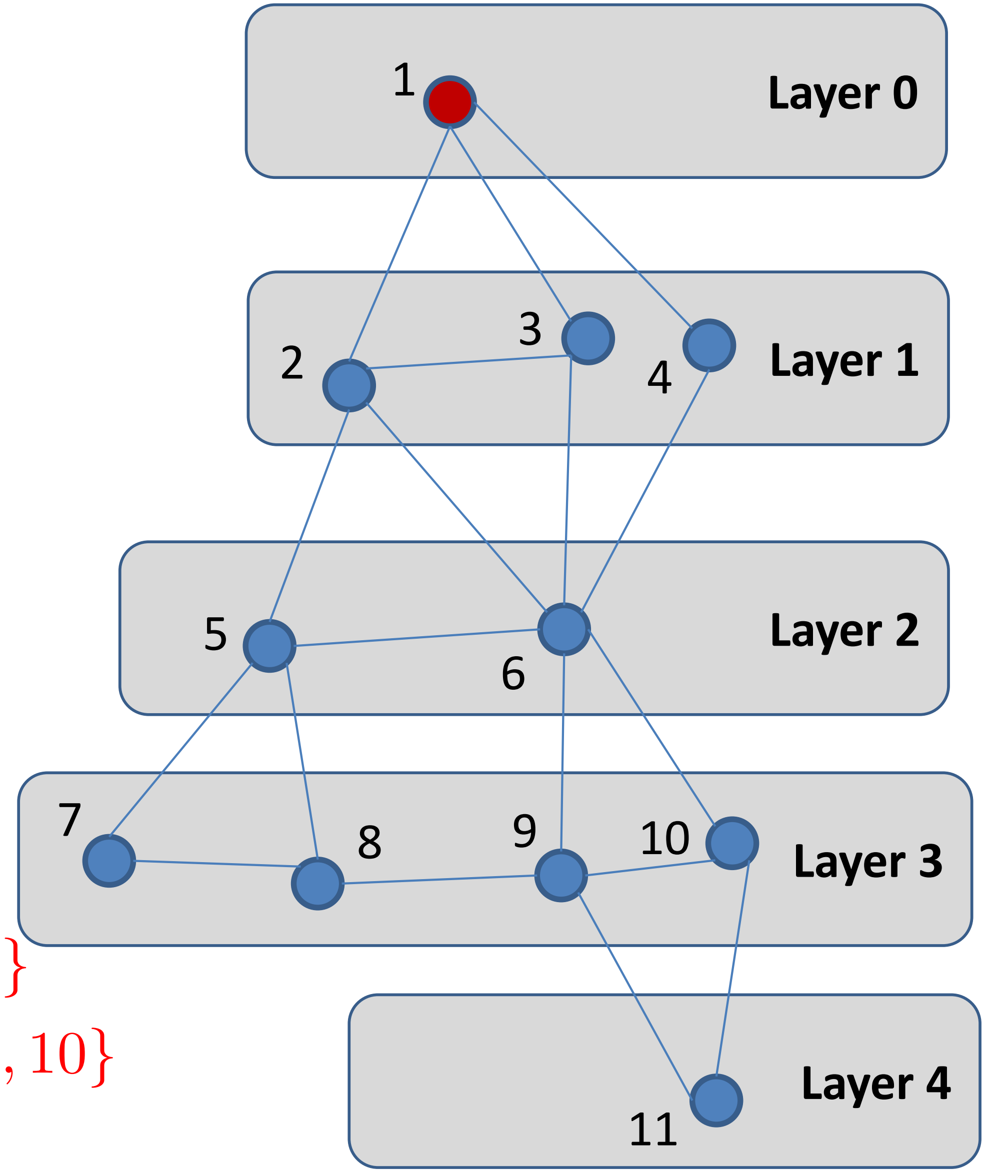
Algorithm:

- $L_0 = \{s\}$
- 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.

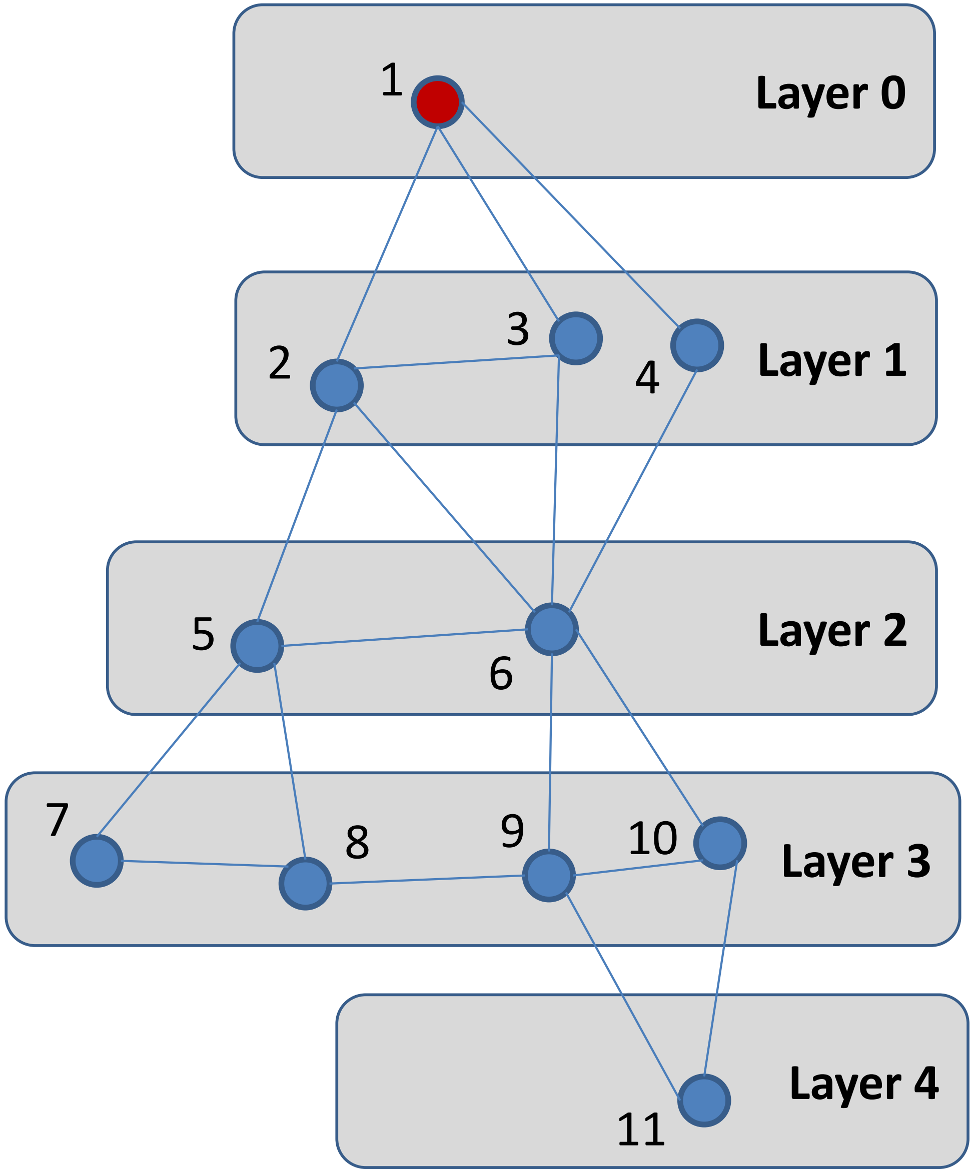
$$
\begin{aligned}
L_0 &= \{1\} \\
L_1 &= N(1) = \{2, 3, 4\} \\
L_2 &= N(L_1) - L_1 - L_0 = \{5, 6, 7\} \\
L_3 &= N(L_2) - L_2 - L_1 = \{7, 8, 9, 10\} \\
L_4 &= N(L_3) - L_3 - L_2 = \{11\}
\end{aligned}
$$



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.



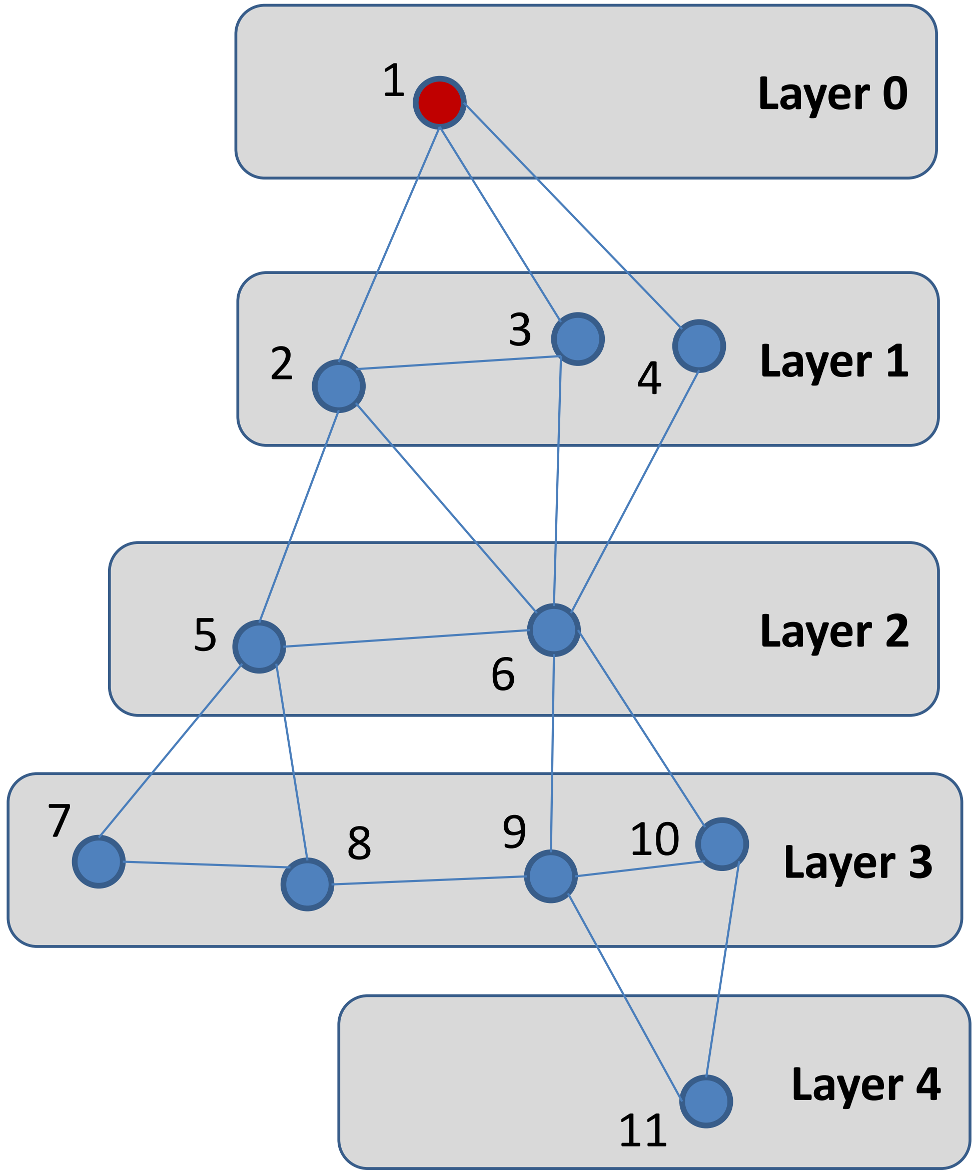
Invariant: each L_i is sorted.

Breadth First Search

Example:

$L_0 = \{1\}$

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



Breadth First Search

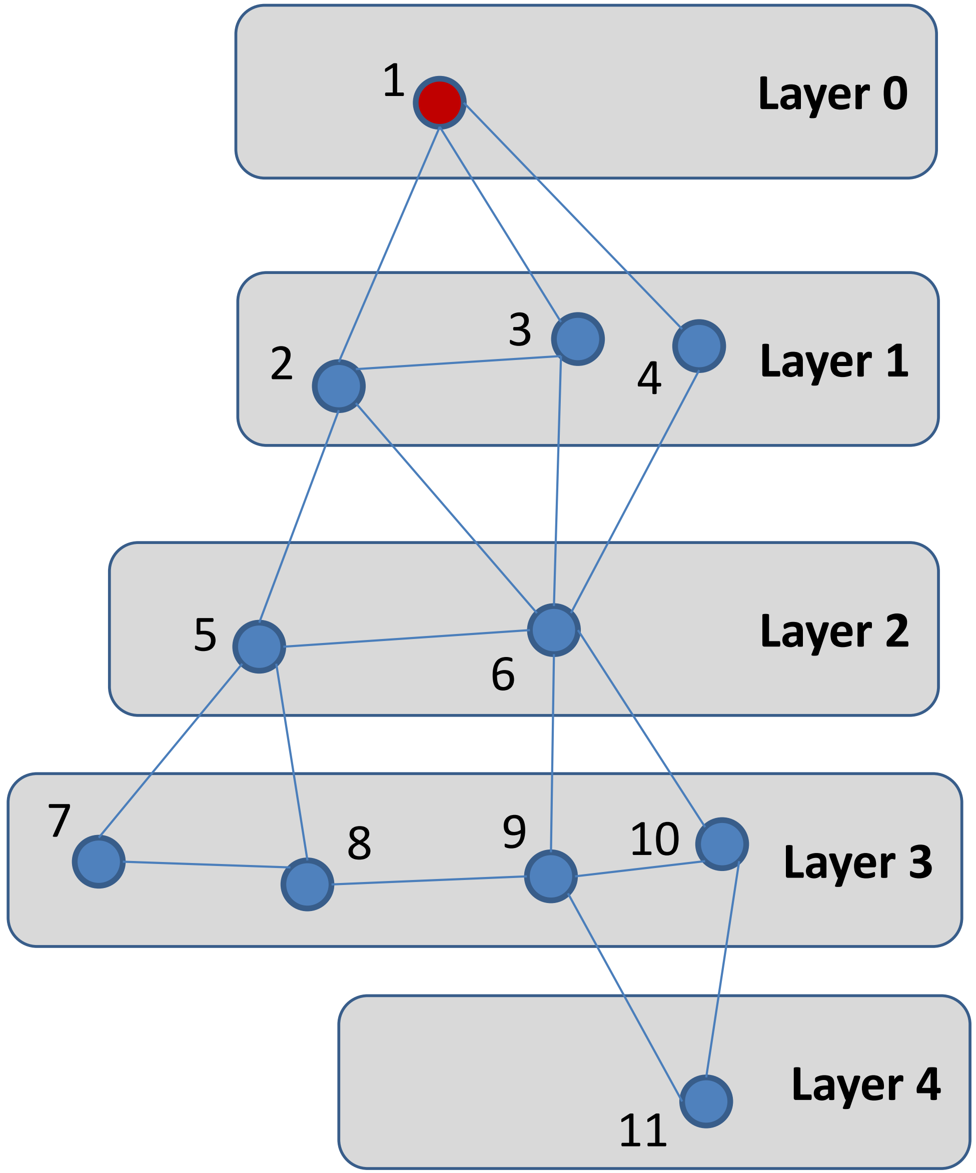
Example:

$L_0 = \{1\}$

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

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

Cost?



Breadth First Search

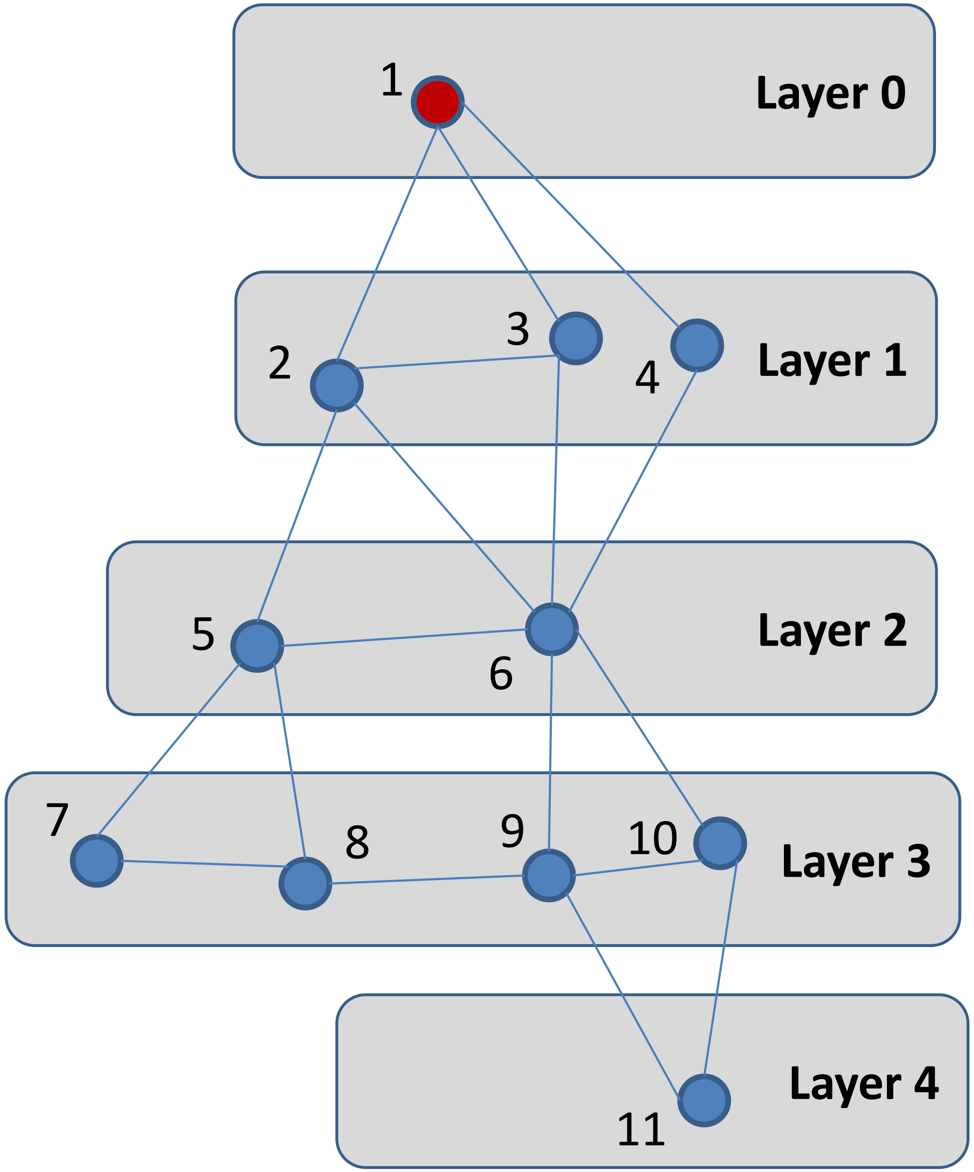
Example:

$L_0 = \{1\}$

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

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

Cost:
$|L_1|/B +$



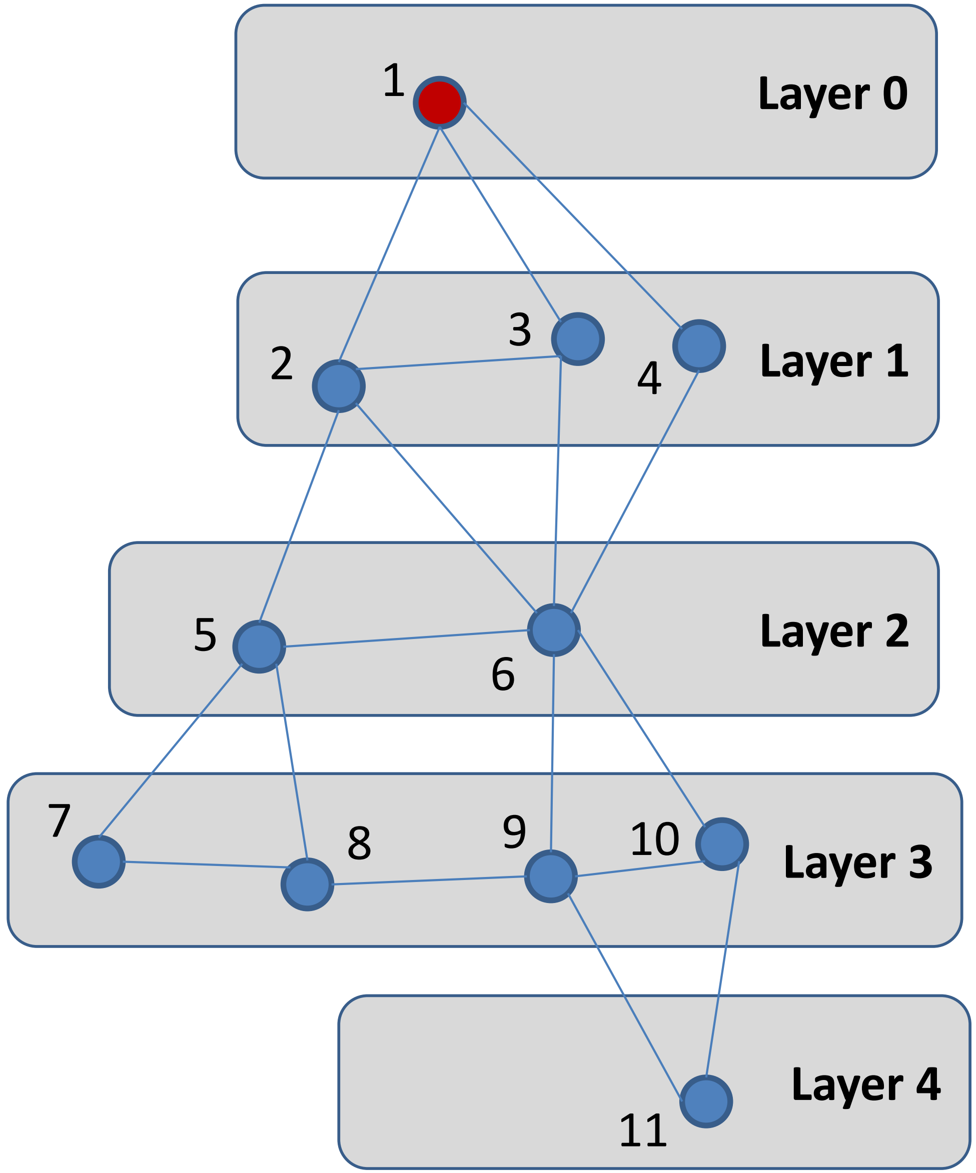
Breadth First Search

Example:

$L_0 = \{1\}$

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

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



Cost:
$|L_1|/B + |L_1| +$

Breadth First Search

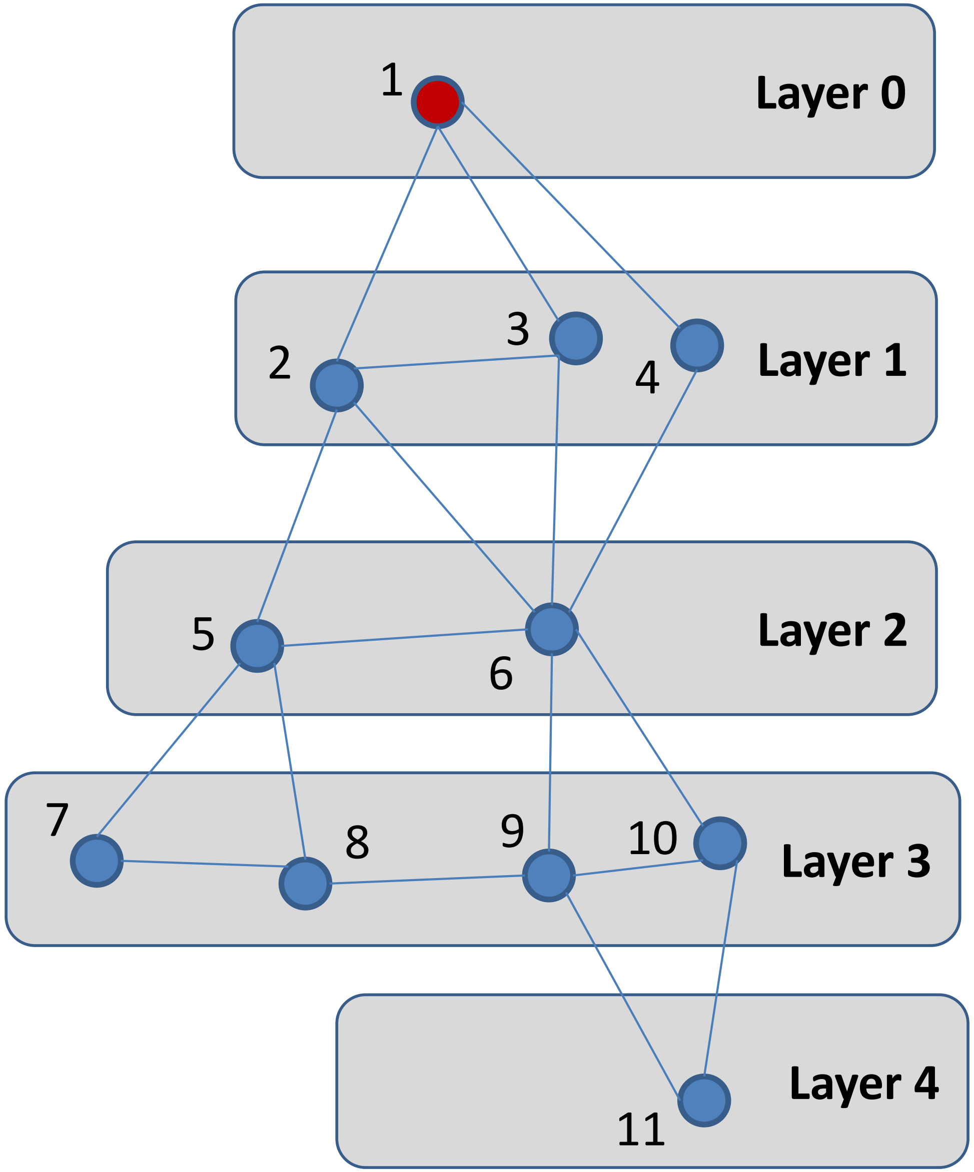
Example:

$L_0 = \{1\}$

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

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

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



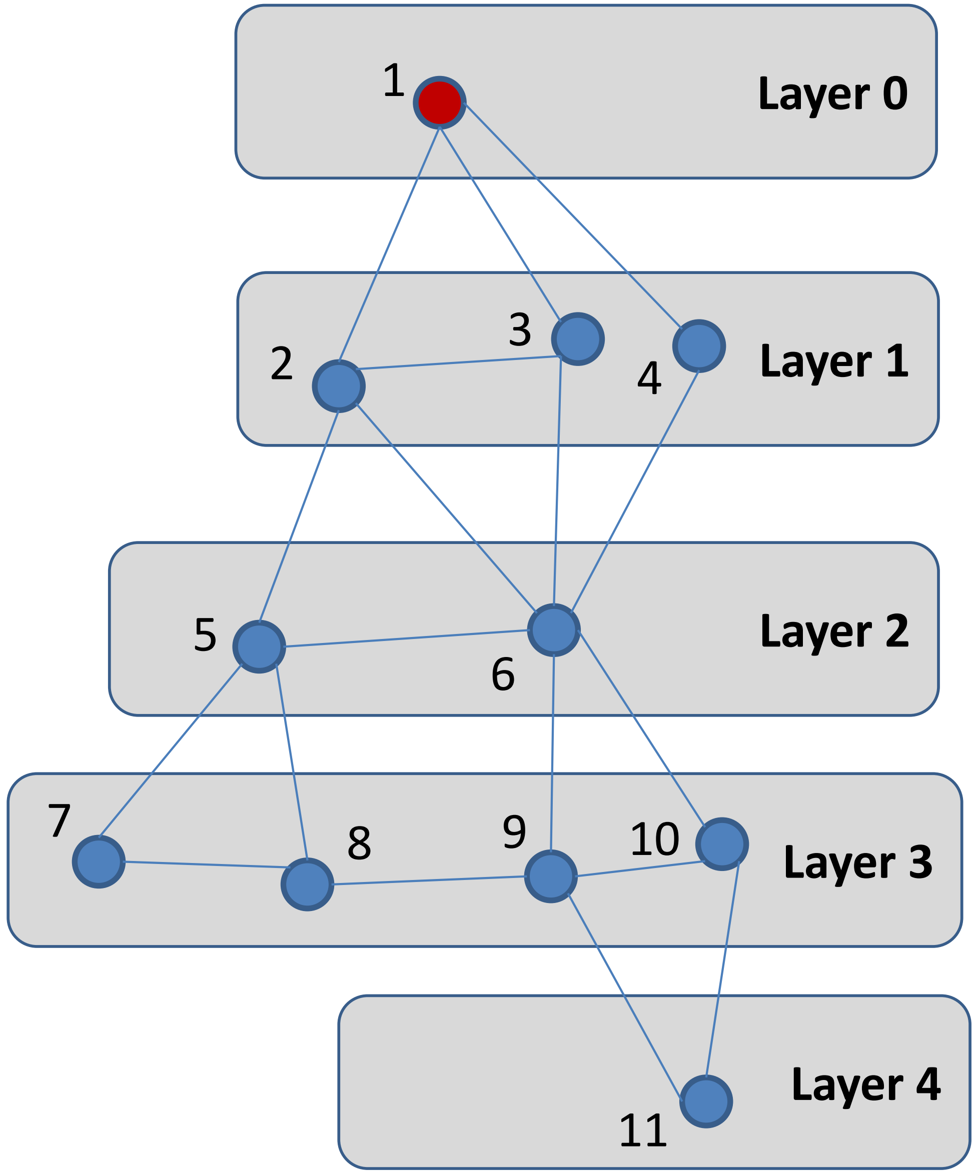
Breadth First Search

Example:

$L_0 = \{1\}$

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

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



Breadth First Search

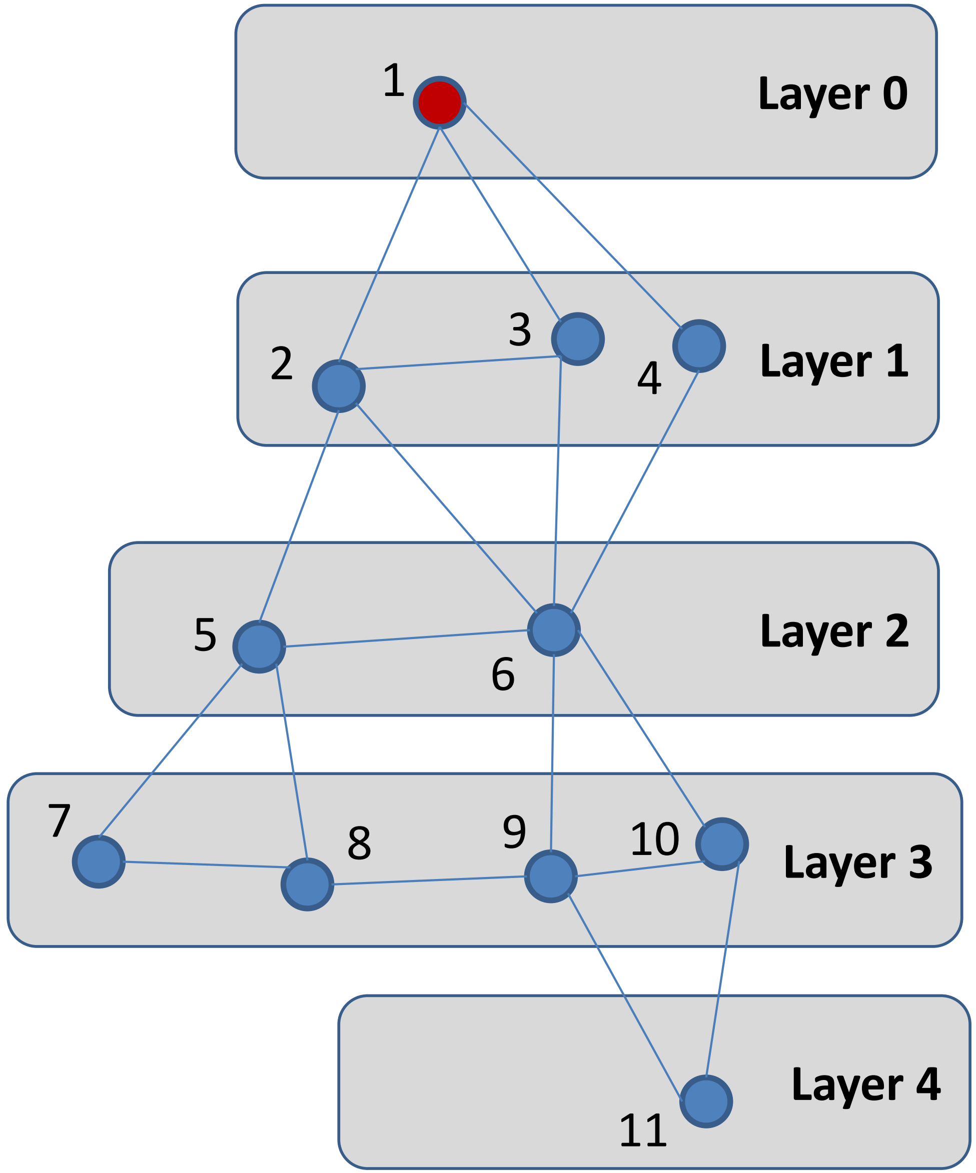
Example:

$L_0 = \{1\}$

Sort

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

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



Breadth First Search

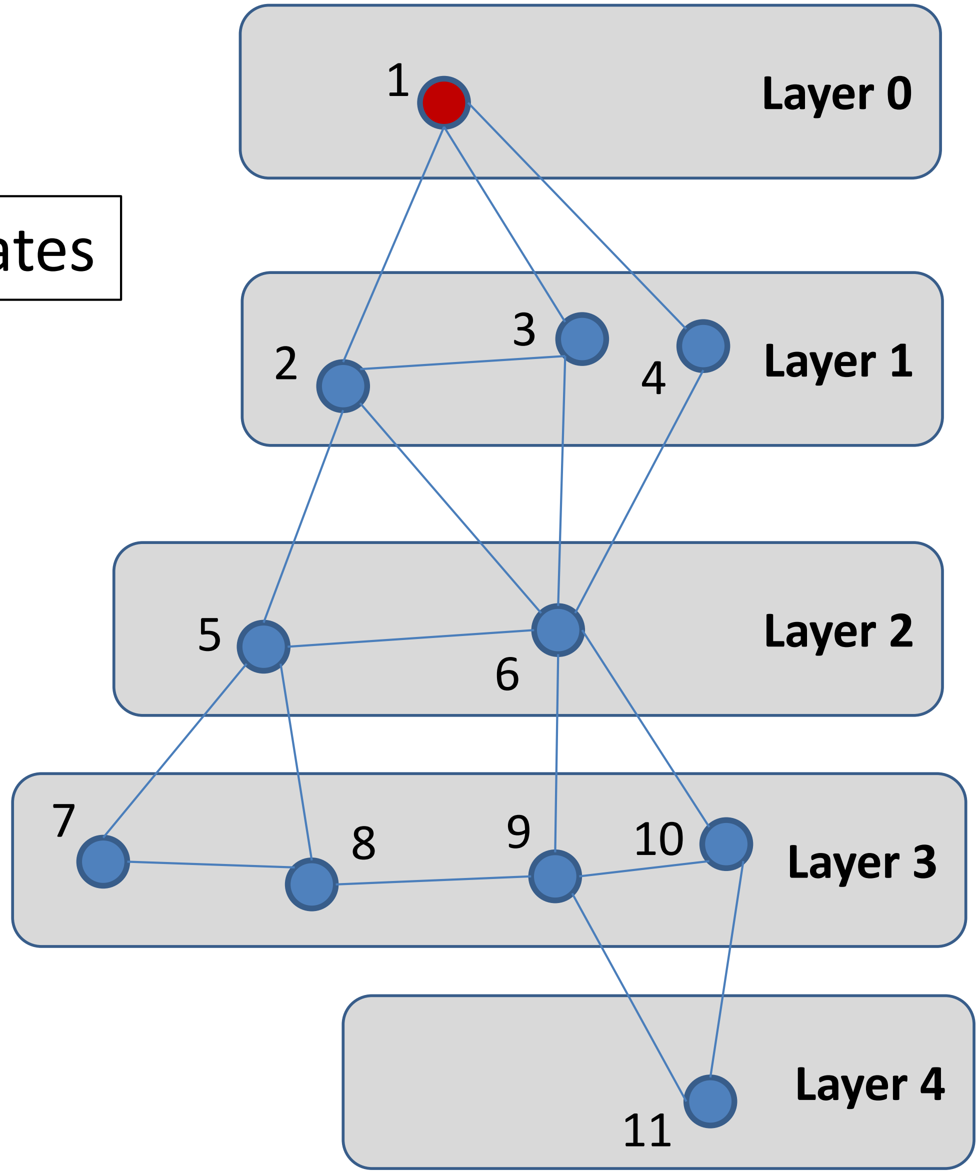
Example:

$L_0 = \{1\}$

Remove duplicates

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

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



Breadth First Search

Example:

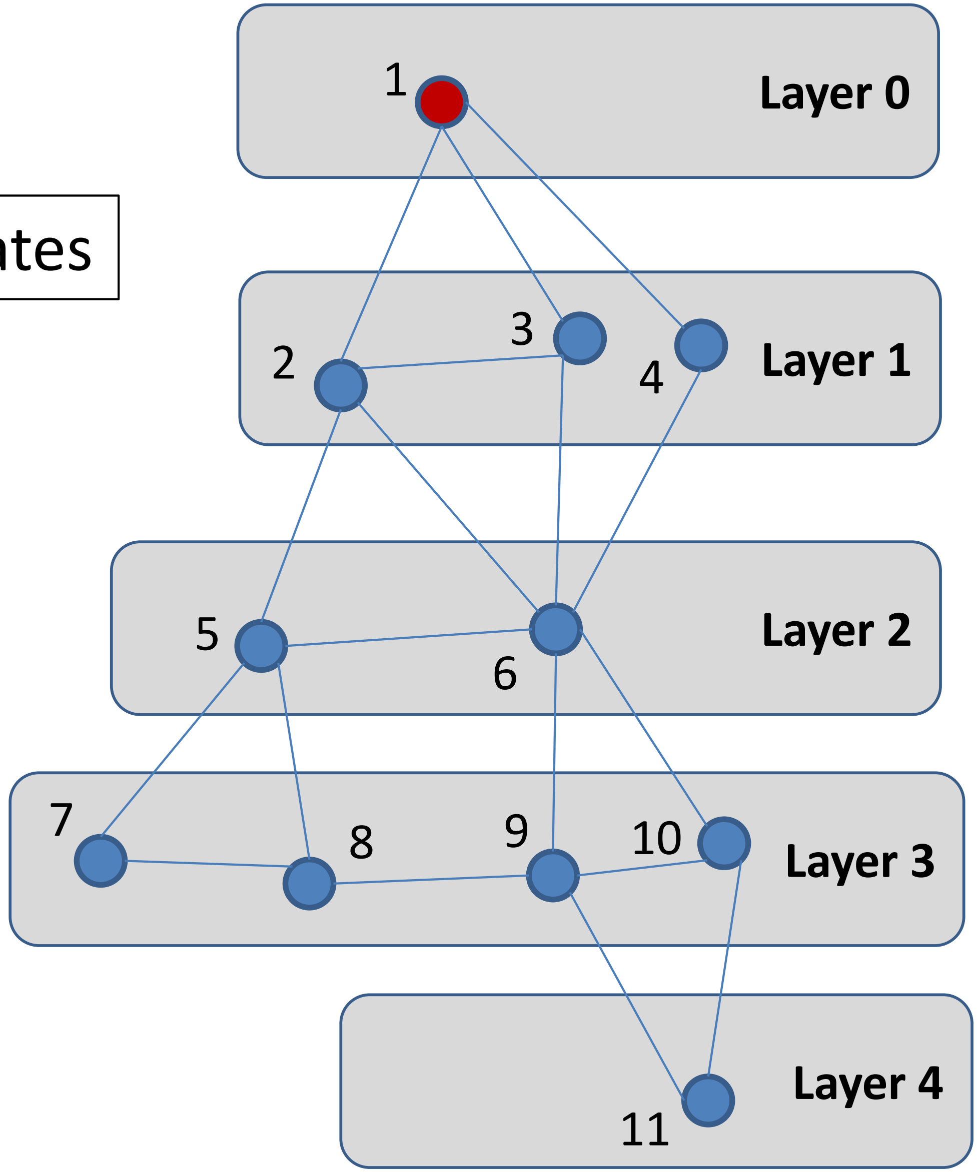
$L_0 = \{1\}$

Remove duplicates

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

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

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



Breadth First Search

Example:

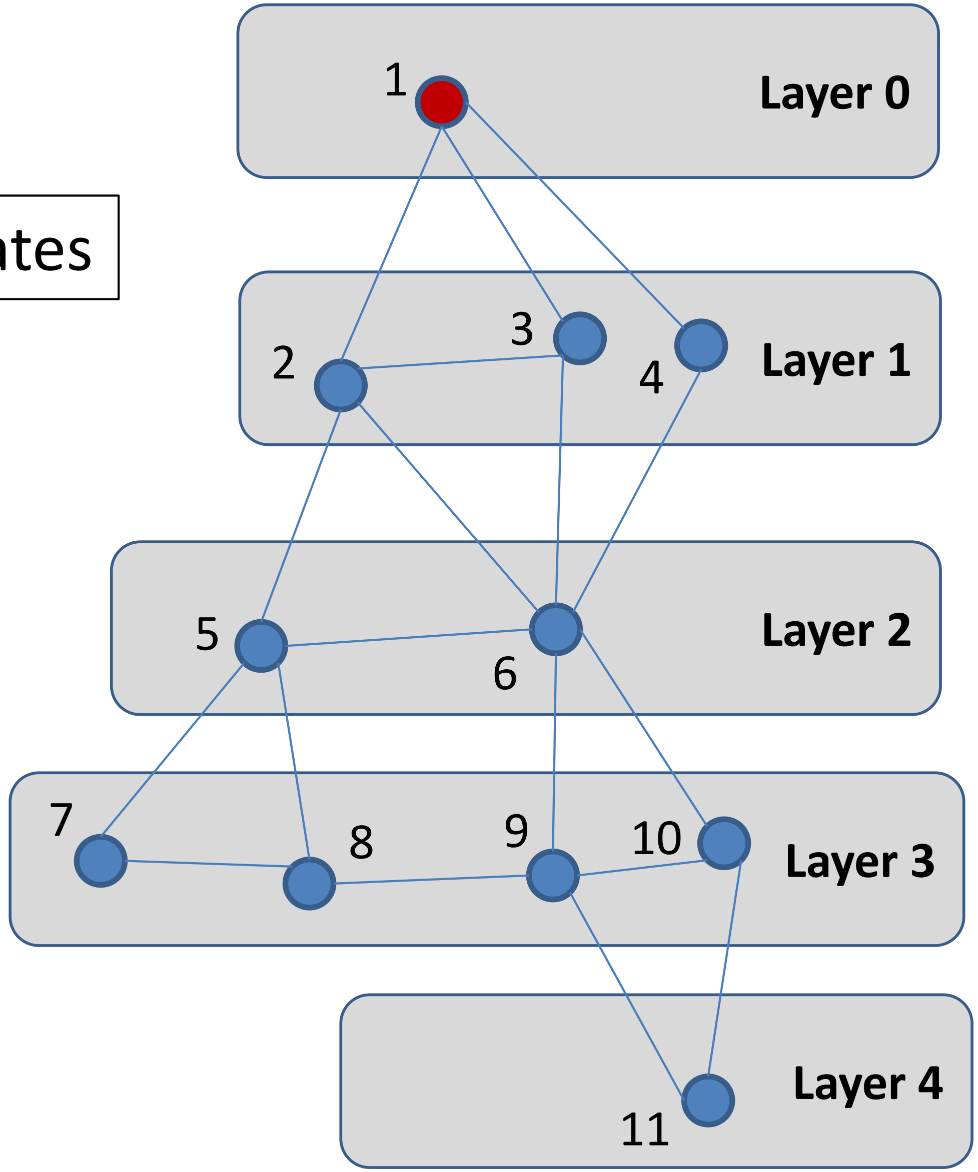
$L_0 = \{1\}$

Remove duplicates

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

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

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



Breadth First Search

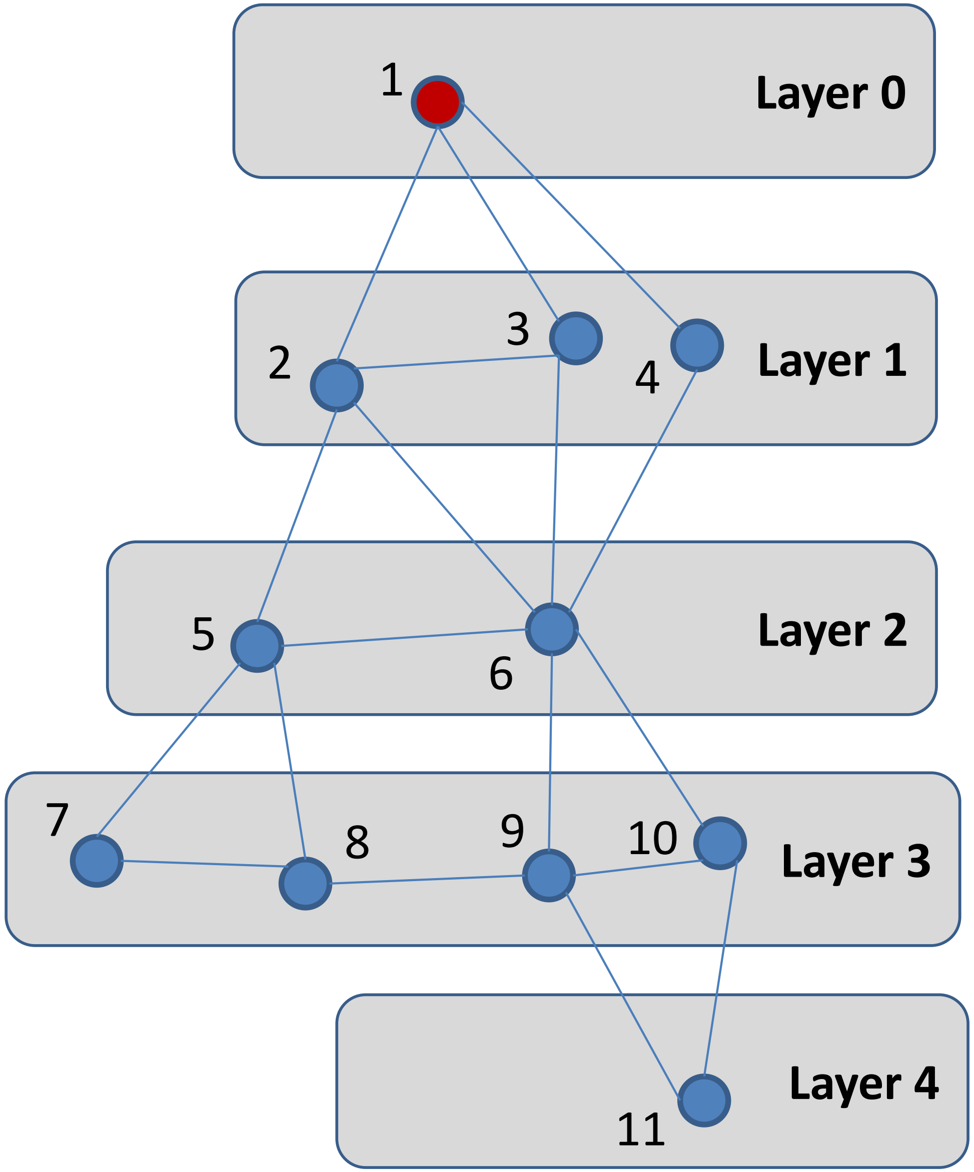
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

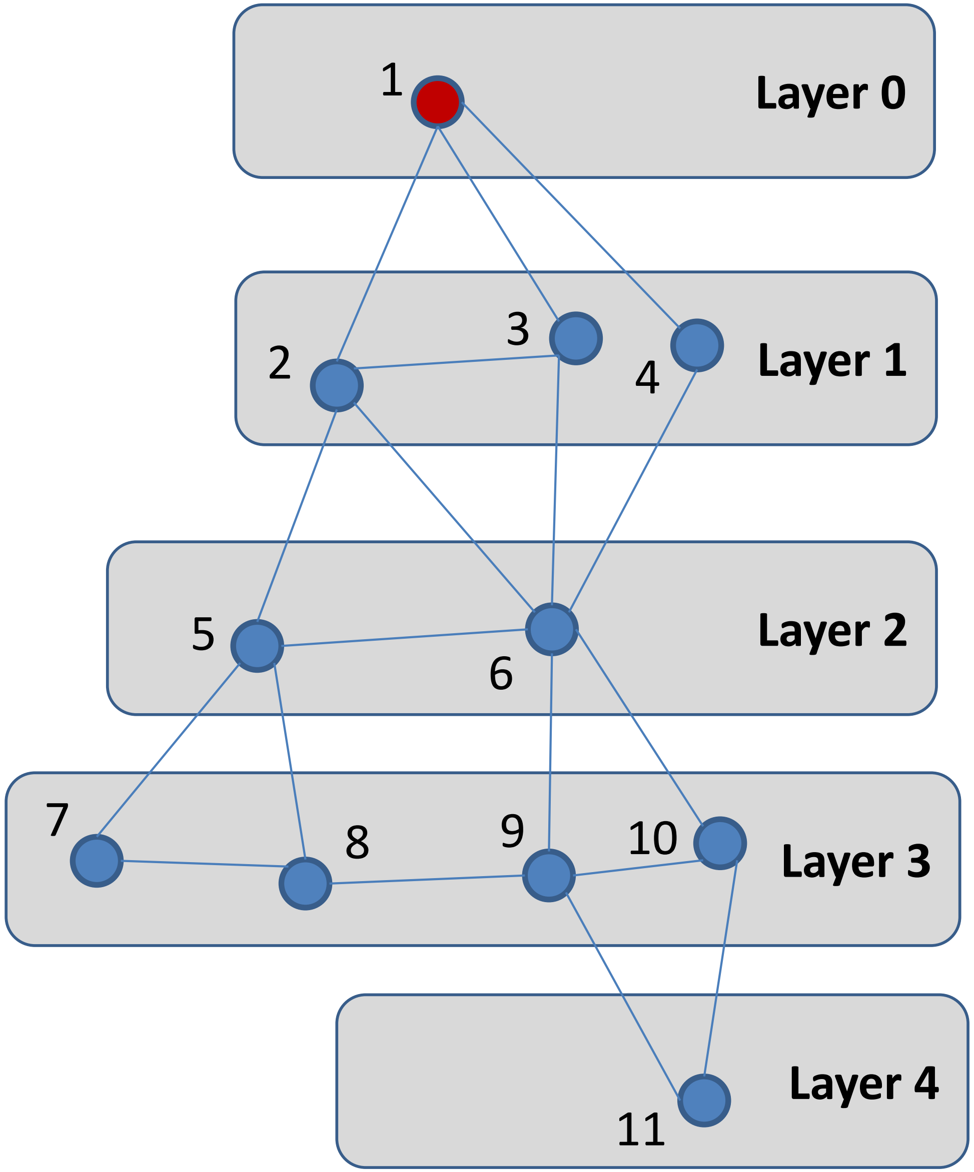
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

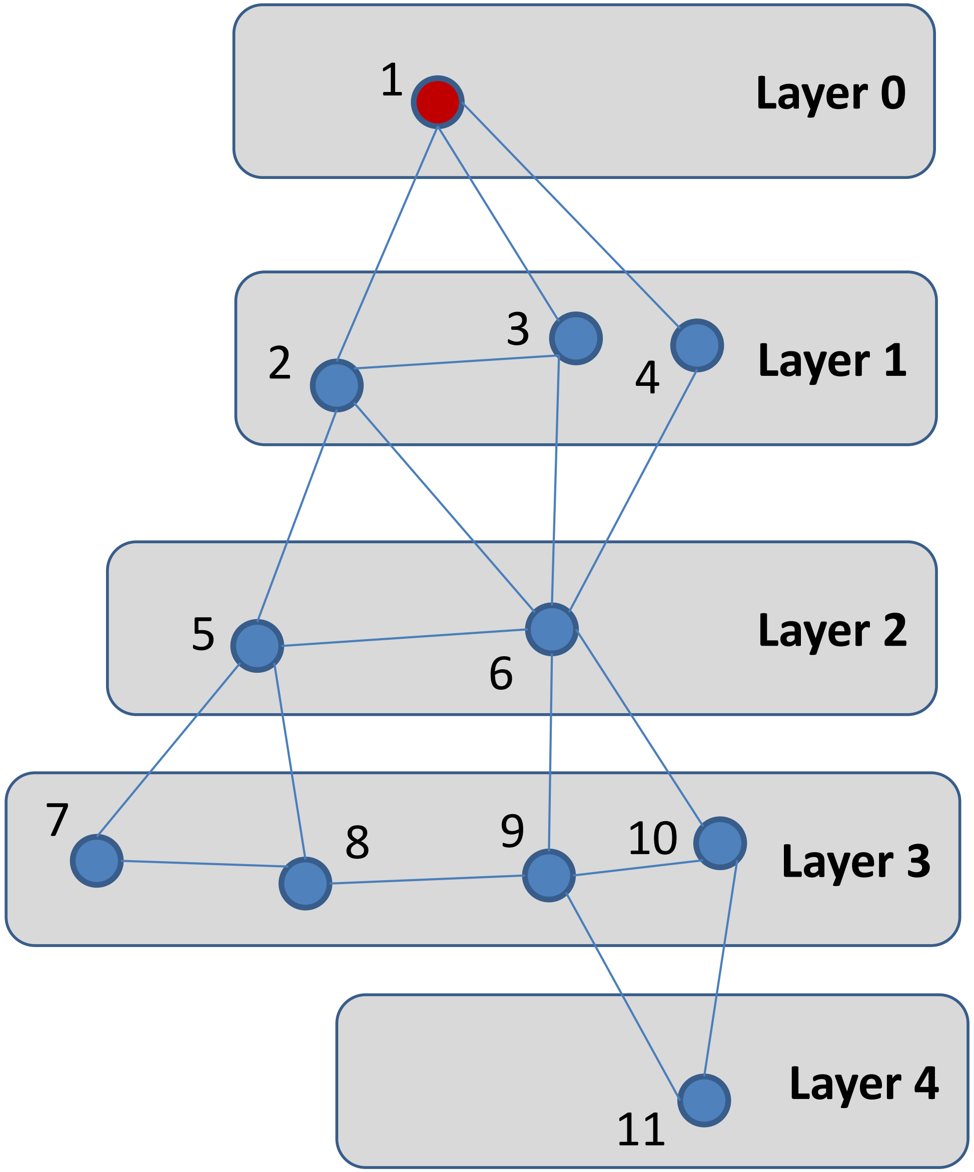
Example:

Subtract L_1.

$L_0 = \{1\}$

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

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



Breadth First Search

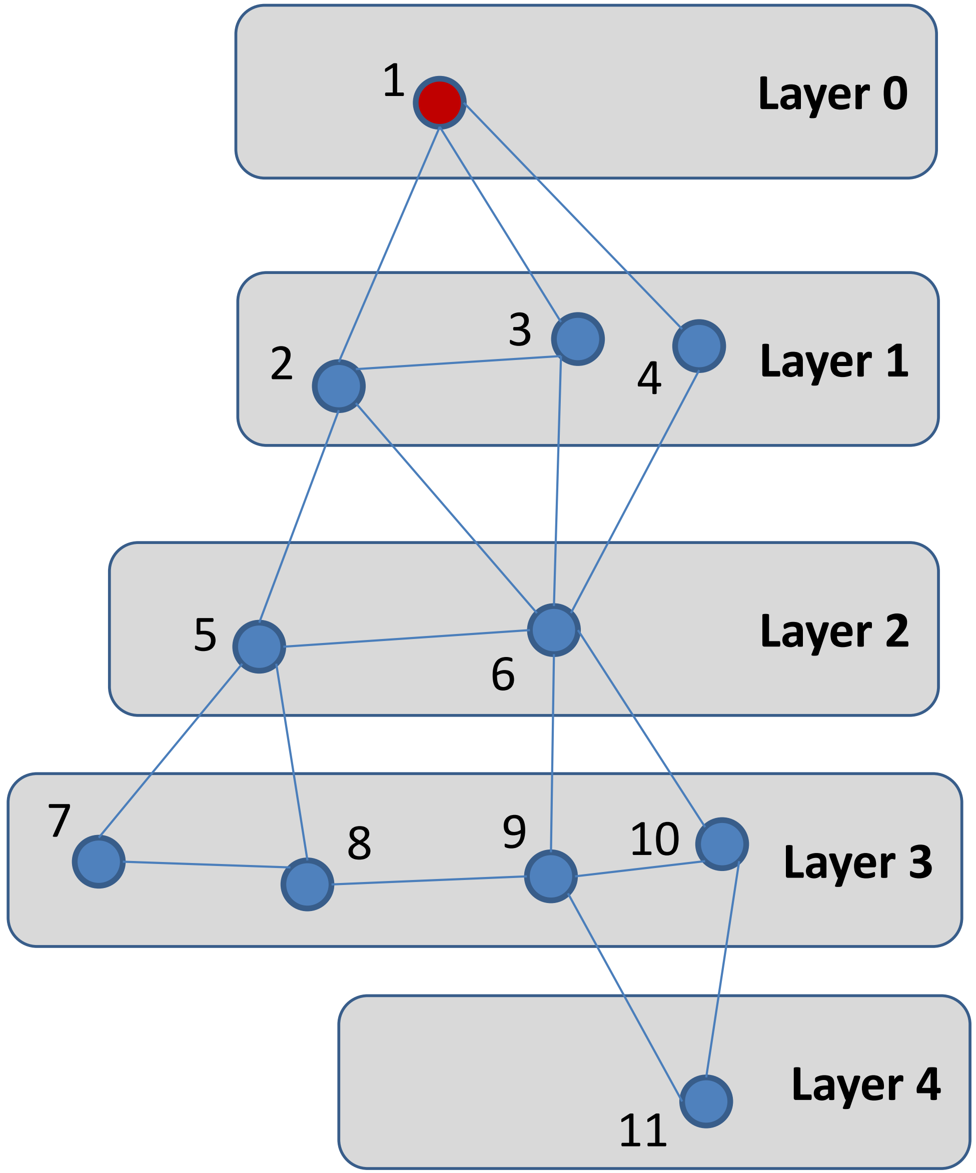
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

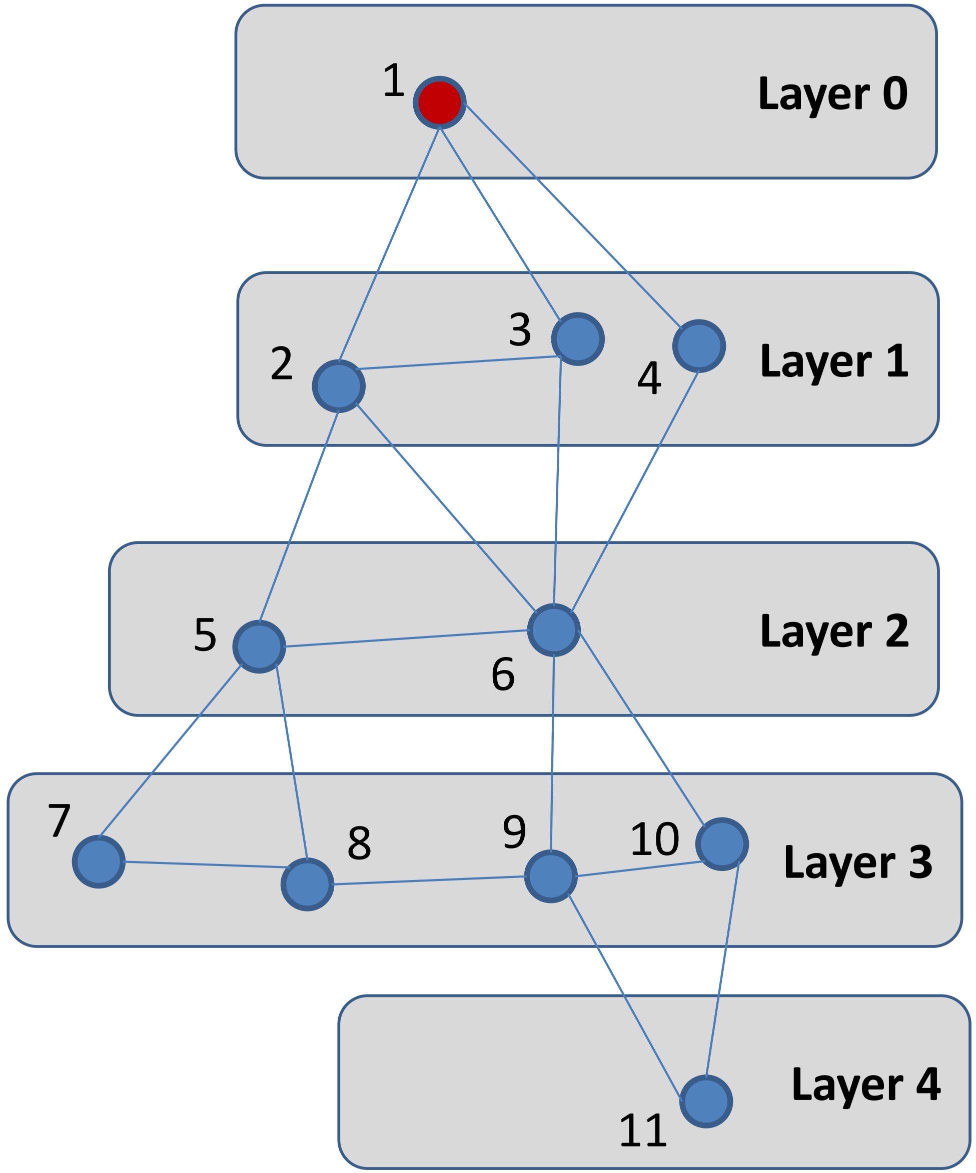
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

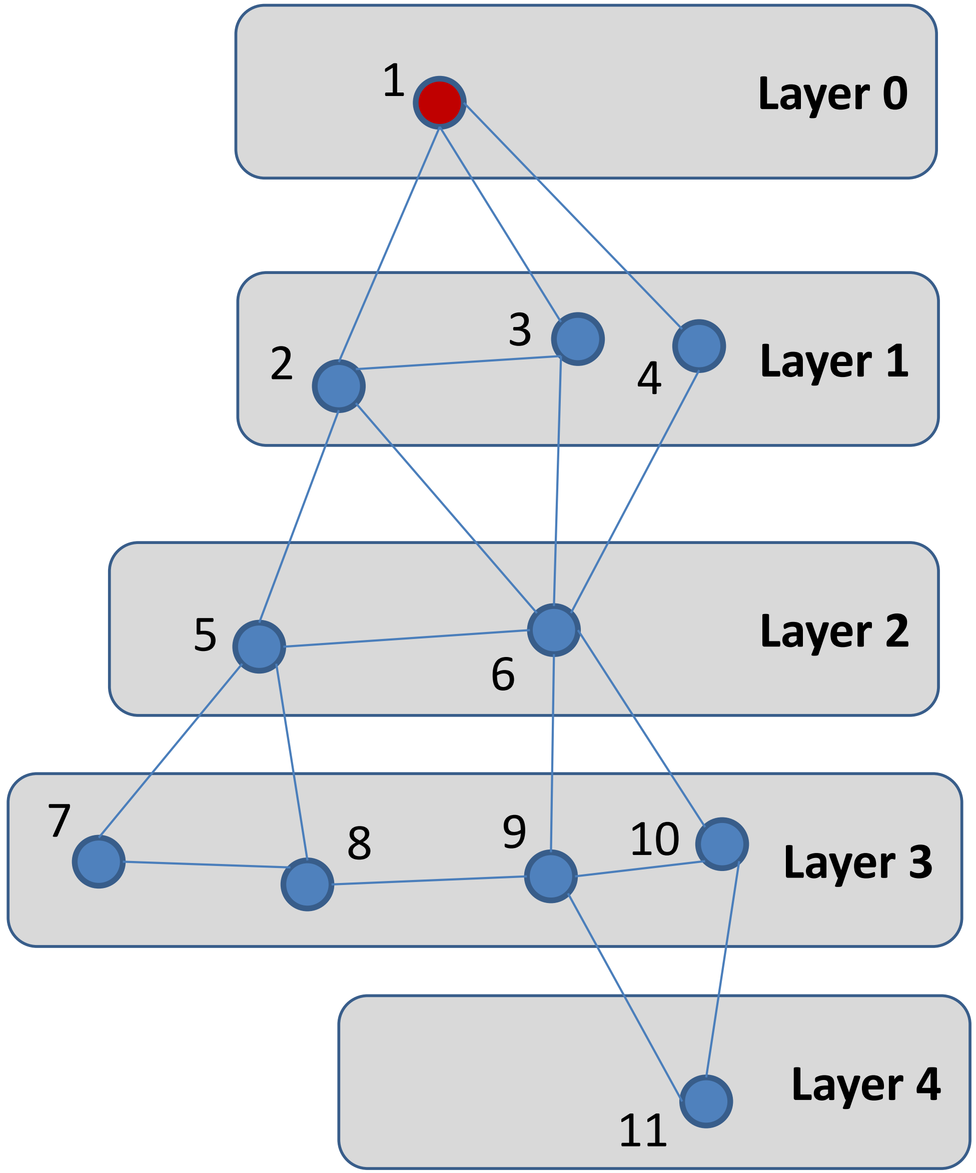
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

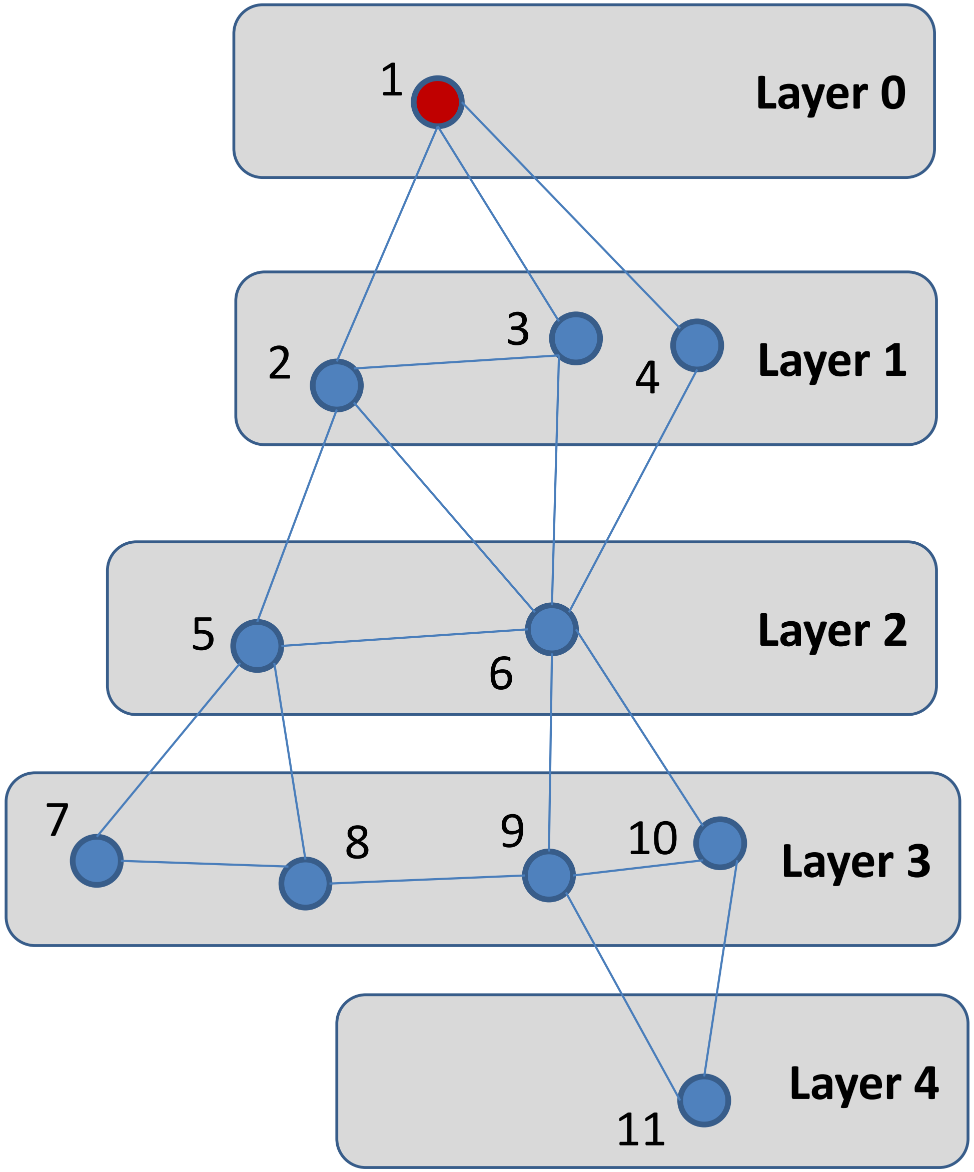
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

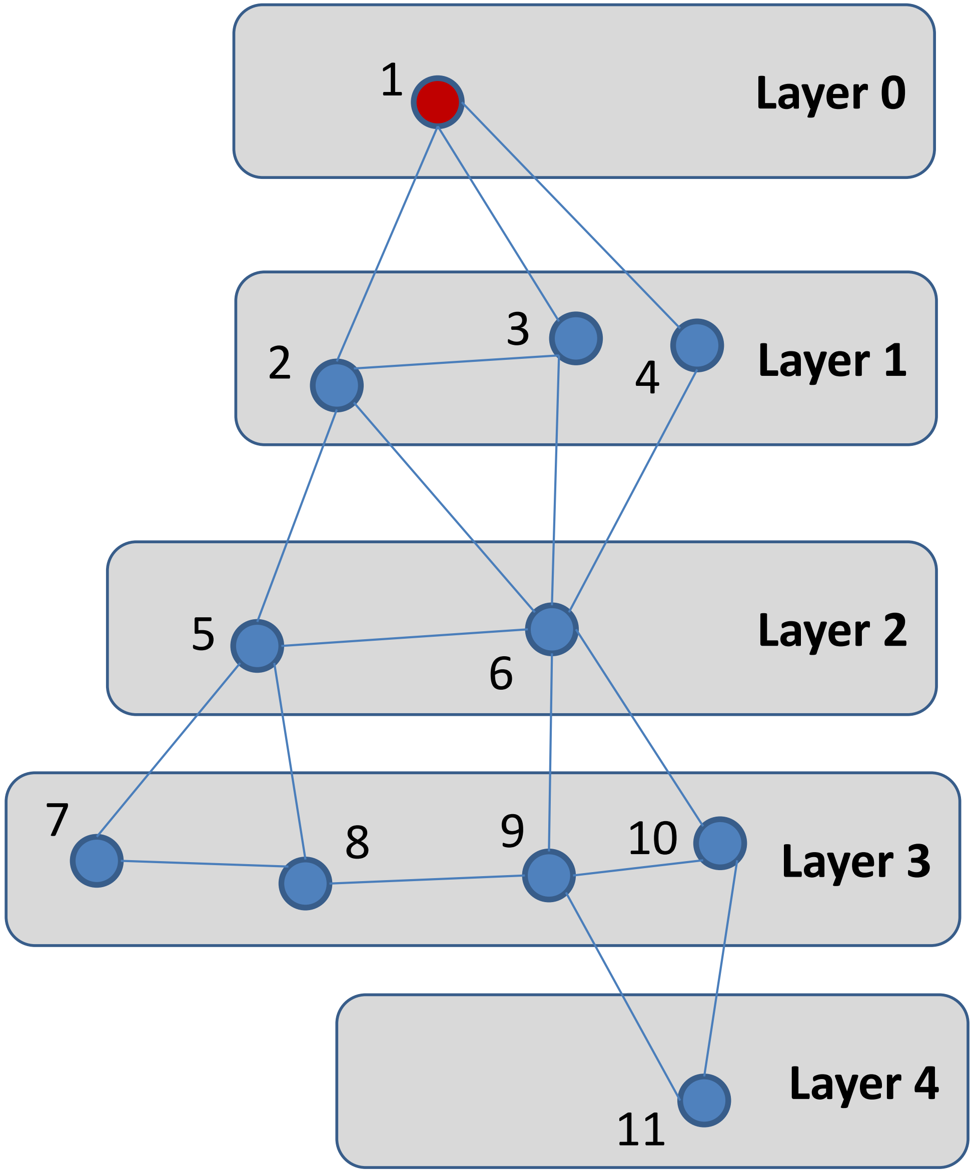
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

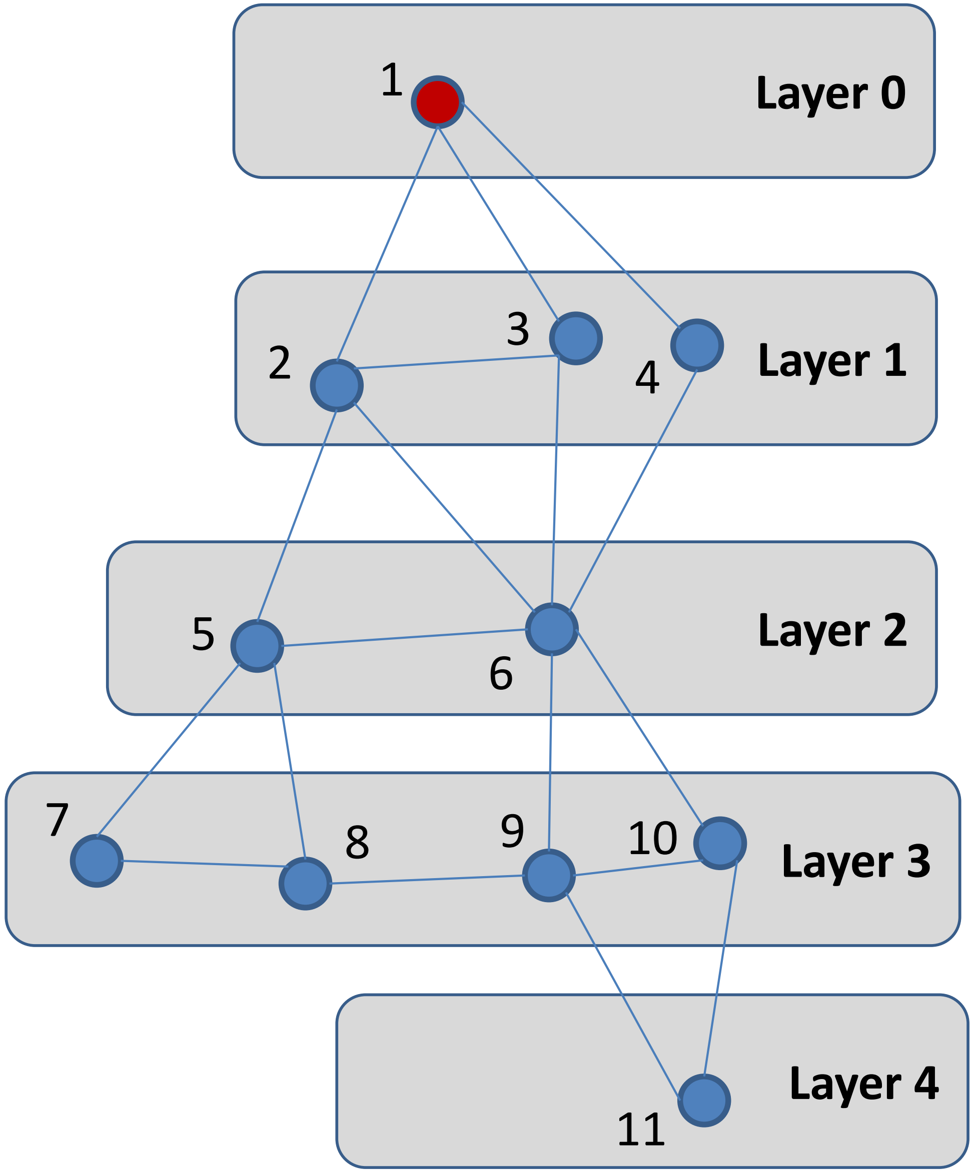
Example:

$L_0 = \{1\}$

Subtract L_1.

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

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



Breadth First Search

Example:

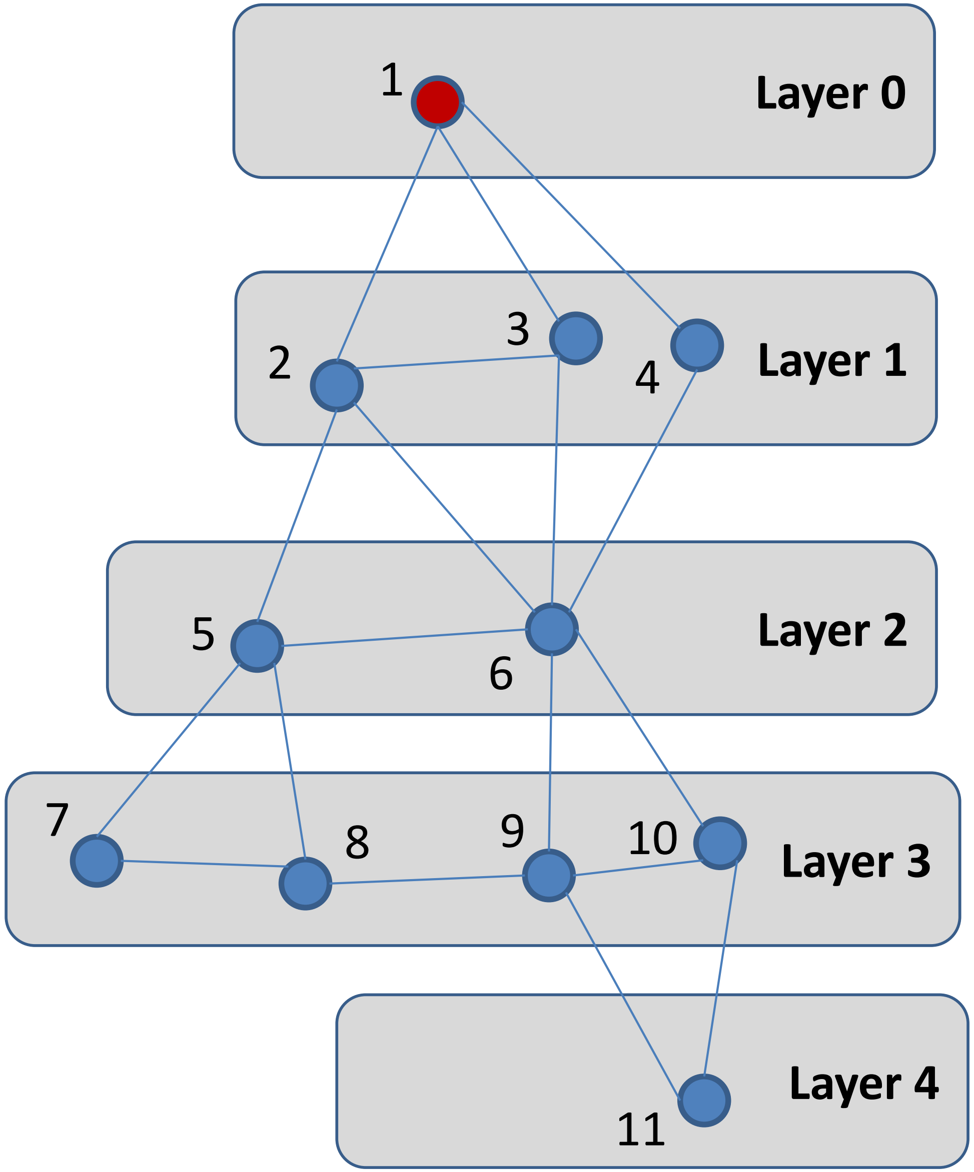
$L_0 = \{1\}$

Subtract L_1.

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

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

$O(|L_1|/B + O(edges(L_1)/B)$



Breadth First Search

Example:

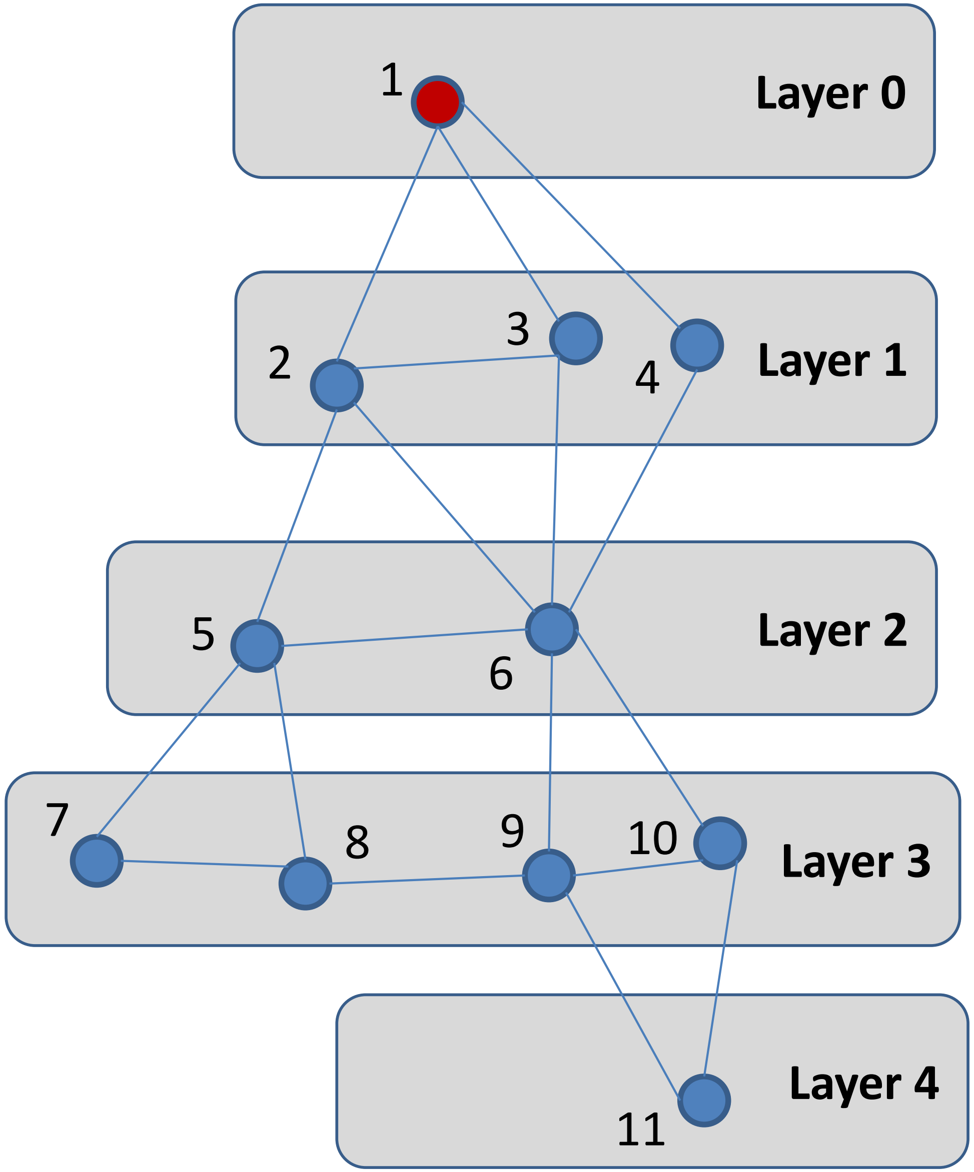
$L_0 = \{1\}$

Subtract L_1.

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

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

$O(|L_1|/B + O(edges(L_1)/B)$



Breadth First Search

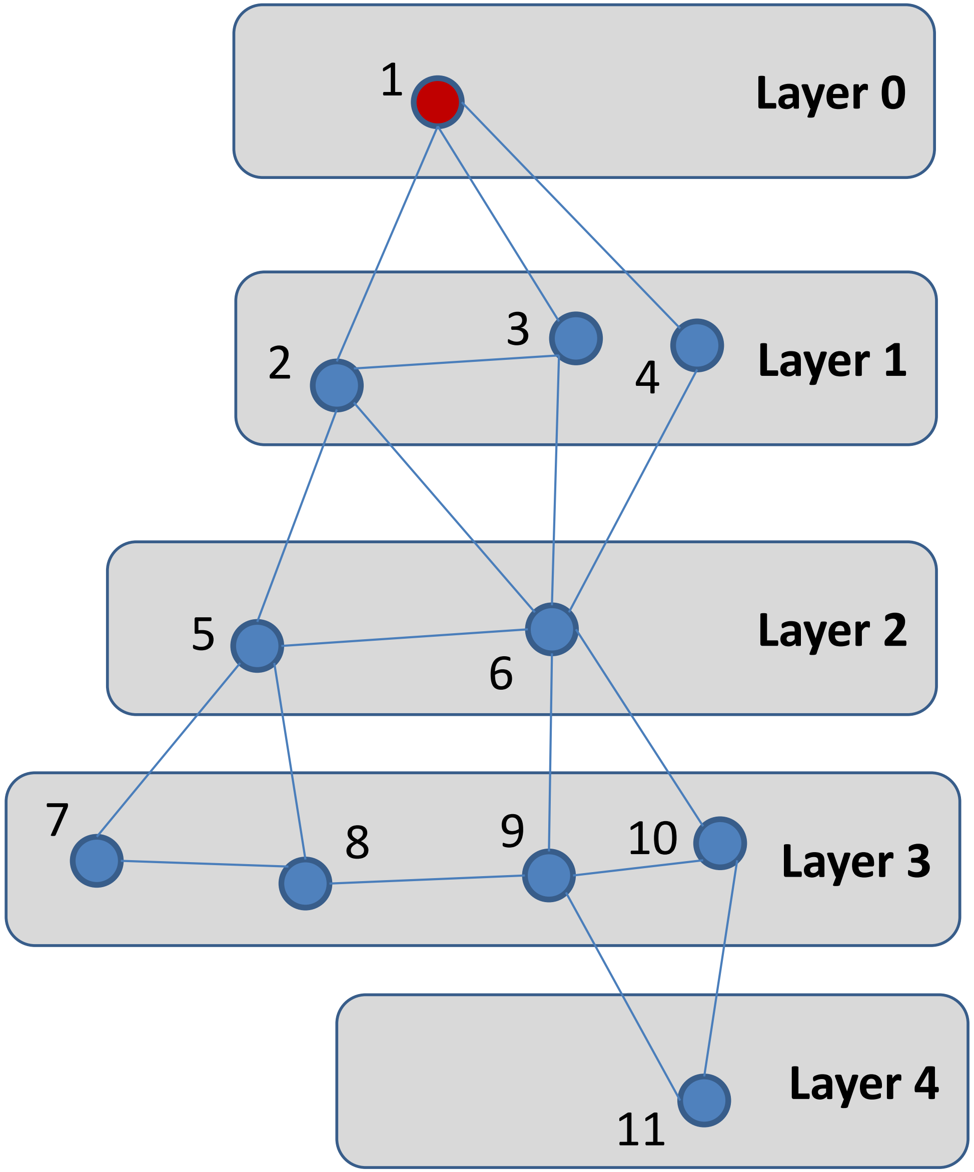
Example:

$L_0 = \{1\}$

Subtract L_0.

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

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



Breadth First Search

Example:

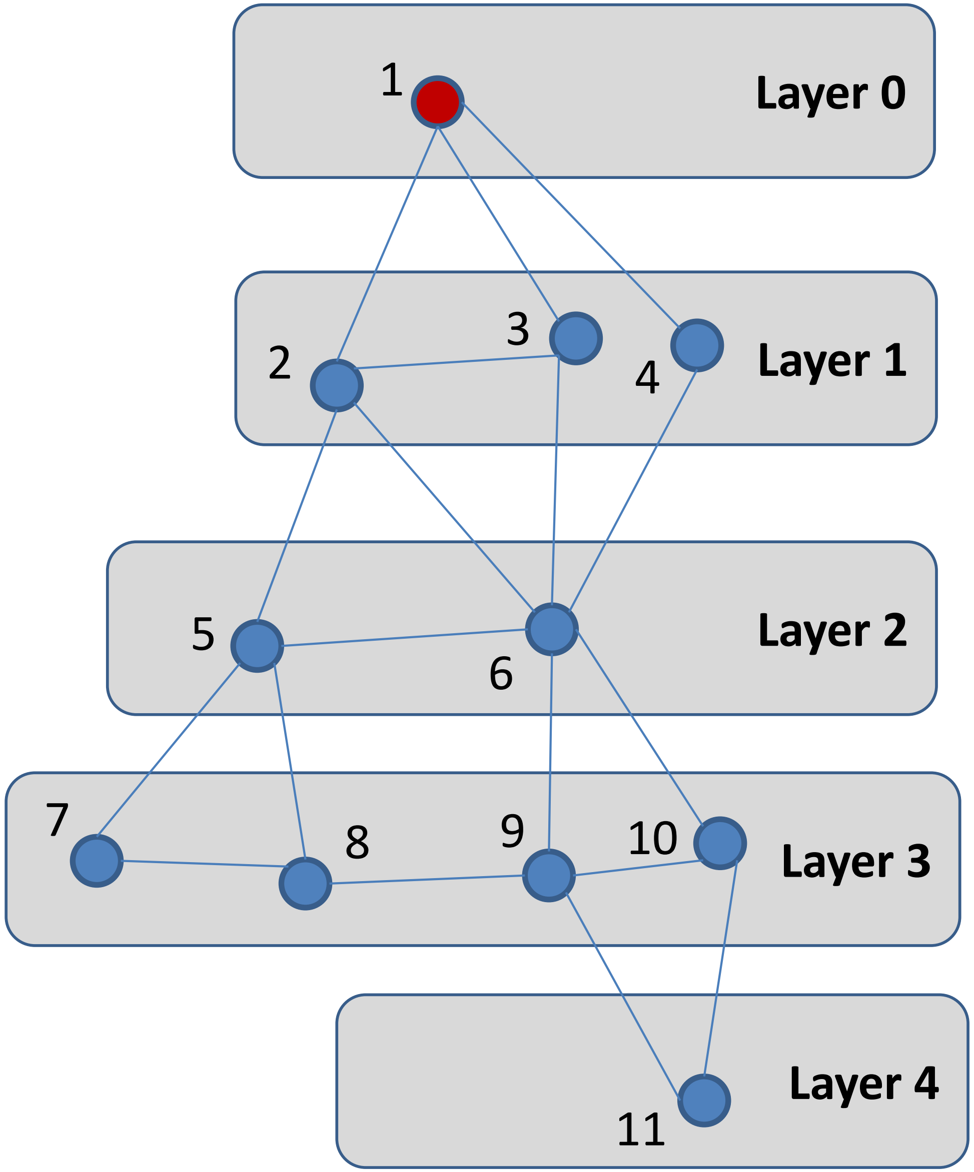
$L_0 = \{1\}$

Subtract L_0.

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

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

$O(|L_0|/B + O(edges(L_1)/B))$



Breadth First Search

Example:

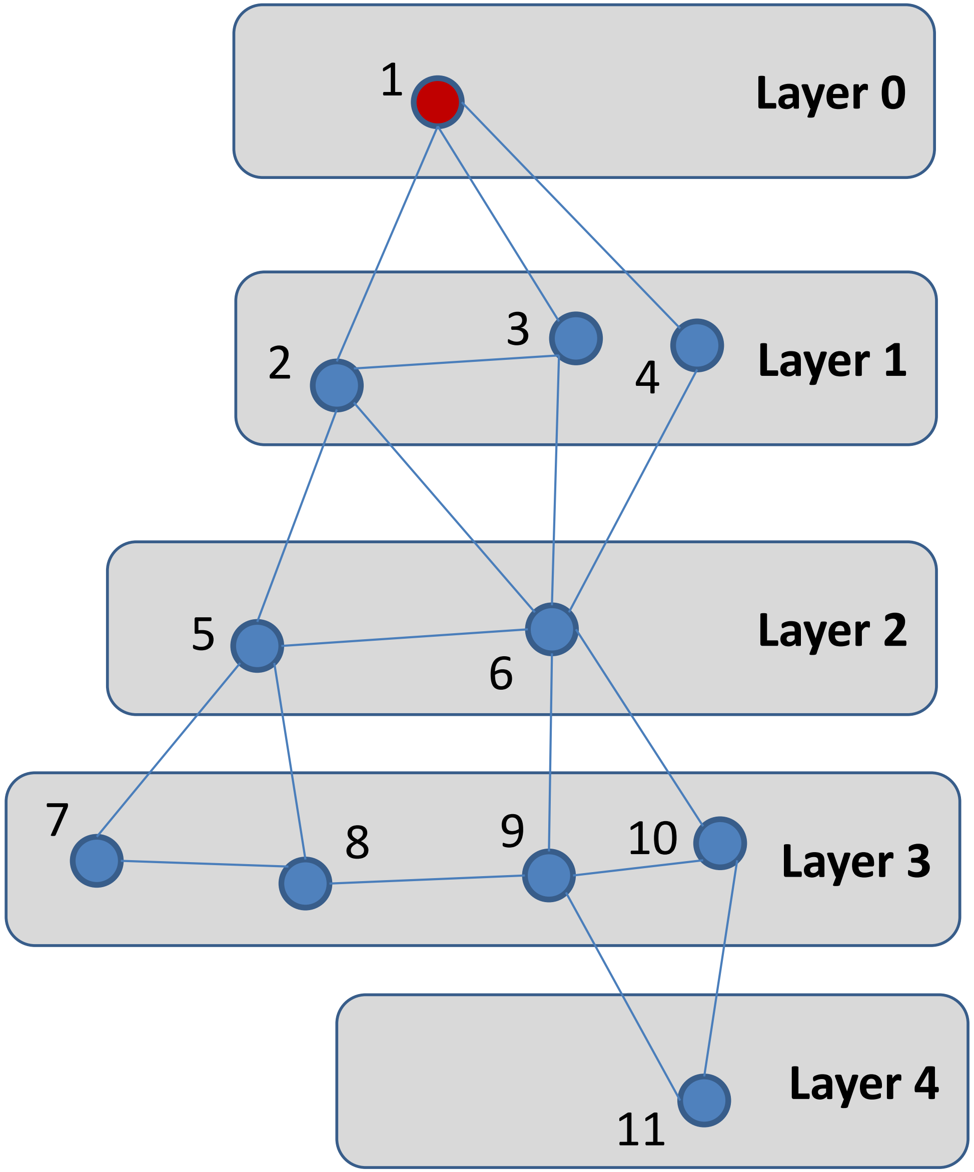
$L_0 = \{1\}$

Subtract L_0.

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

$L_2 = \{5, 6\}$

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



Breadth First Search

Cost to construct L_{i+1} :

1. L_{i+1} = neighbors of all nodes in L_i $\quad 2|L_i| + edges(L_i)/B$
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 $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. $sort(L_i)$
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 — $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
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 — $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
4. Scan L_i, L_{i+1}: remove nodes in both. — $|L_i|/B + edges(L_i)/B$
5. Scan L_{i-1}, L_{i+1}: remove nodes in both. — $|L_{i-1}|/B + edges(L_i)/B$

Breadth First Search

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

Cost to construct L_{i+1} :

1. L_{i+1} = neighbors of all nodes in L_i — $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
4. Scan L_i, L_{i+1}: remove nodes in both. — $|L_i|/B + edges(L_i)/B$
5. Scan L_{i-1}, L_{i+1}: remove nodes in both. — $|L_{i-1}|/B + edges(L_i)/B$

Breadth First Search

Cost to construct L_{i+1} :

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

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

Sums to 2|E|/B over all levels.

2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
4. Scan L_i, L_{i+1}: remove nodes in both. — $|L_i|/B + edges(L_i)/B$
5. Scan L_{i-1}, L_{i+1}: remove nodes in both. — $|L_{i-1}|/B + edges(L_i)/B$

Breadth First Search

Cost to construct L_{i+1} :

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

1. L_{i+1} = neighbors of all nodes in L_i — $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
4. Scan L_i, L_{i+1}: remove nodes in both. — $|L_i|/B + edges(L_i)/B$
5. Scan L_{i-1}, L_{i+1}: remove nodes in both. — $|L_{i-1}|/B + edges(L_i)/B$

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

Total cost:

$$O(|V| + |E|/B + sort(|E|))$$

First Search

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

1. L_{i+1} = neighbors of all nodes in L_i — $2|L_i| + edges(L_i)/B$
2. Sort L_{i+1}. — $sort(L_i)$
3. Remove duplicates in L_{i+1}. — $edges(L_i)/B$
4. Scan L_i, L_{i+1}: remove nodes in both. — $|L_i|/B + edges(L_i)/B$
5. Scan L_{i-1}, L_{i+1}: remove nodes in both. — $|L_{i-1}|/B + edges(L_i)/B$

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

Sums to 2|V|/B over all levels.

First Search

Total cost:

$O(|V| + |E|/B + sort(|E|))$

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

$2|L_i| + edges(L_i)/B$

Sums to 8|E|/B over all levels

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

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.

$|L_i|/B + edges(L_i)/B$

$|L_{i-1}|/B + edges(L_i)/B$

Sums to 2|V|/B over all levels.

First Search

Total cost:

$$O(|V| + |E|/B + sort(|E|))$$

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

1. L_{i+1} = neighbors of all nodes in L_i

$$2|L_i| + edges(L_i)/B$$

Sums to 8|E|/B over all levels

2. Sort L_{i+1}.

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

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

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

$$|L_i|/B + edges(L_i)/B$$

$$L_{i-1}|/B + edges(L_i)/B$$

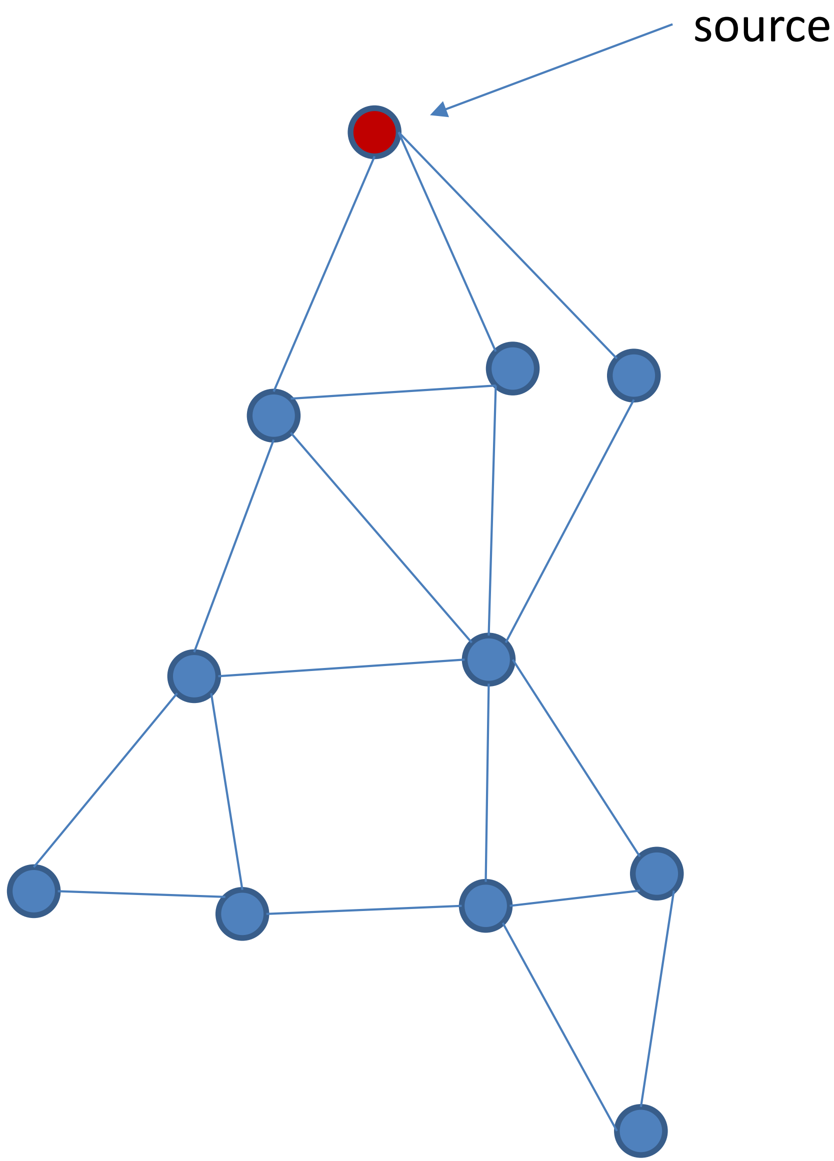
over all levels.

Compare to:

$$O(|V| + |E|)$$

Problem: Breadth First Search

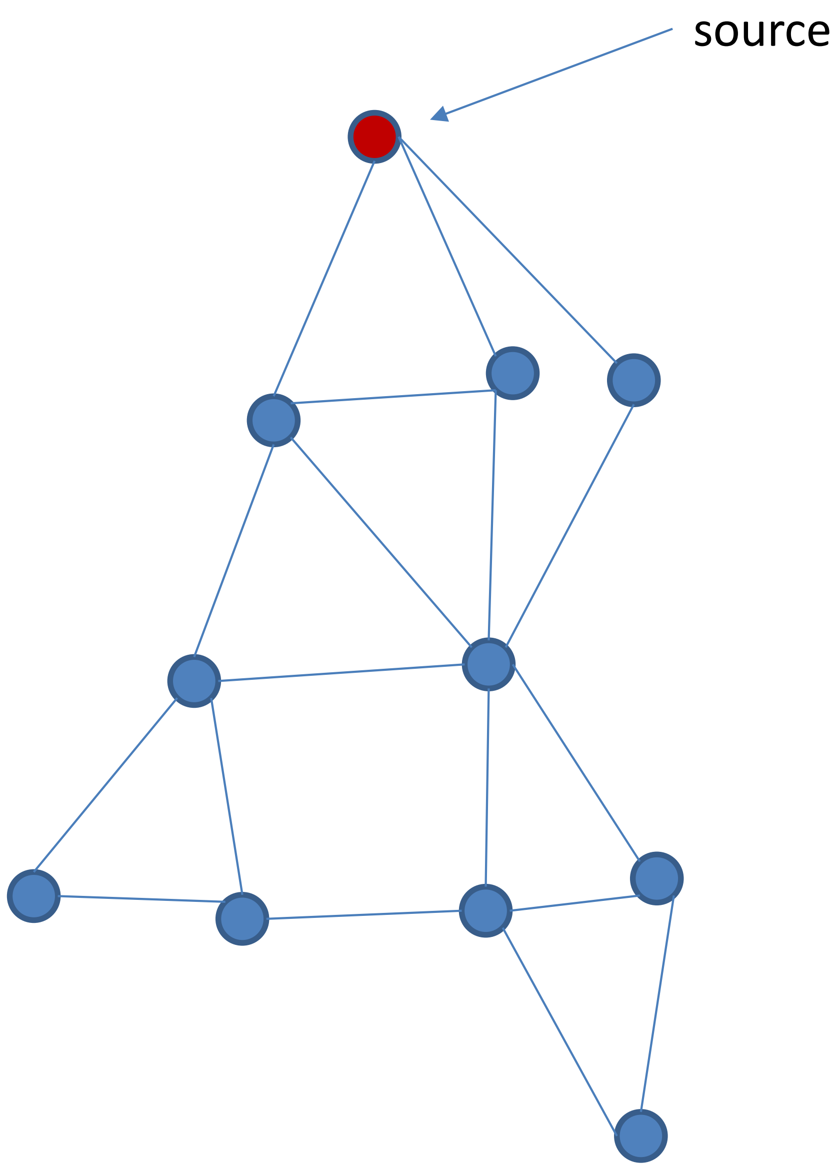
Can we do better?



Problem: Breadth First Search

Can we do better?

Unlikely in dense graph.

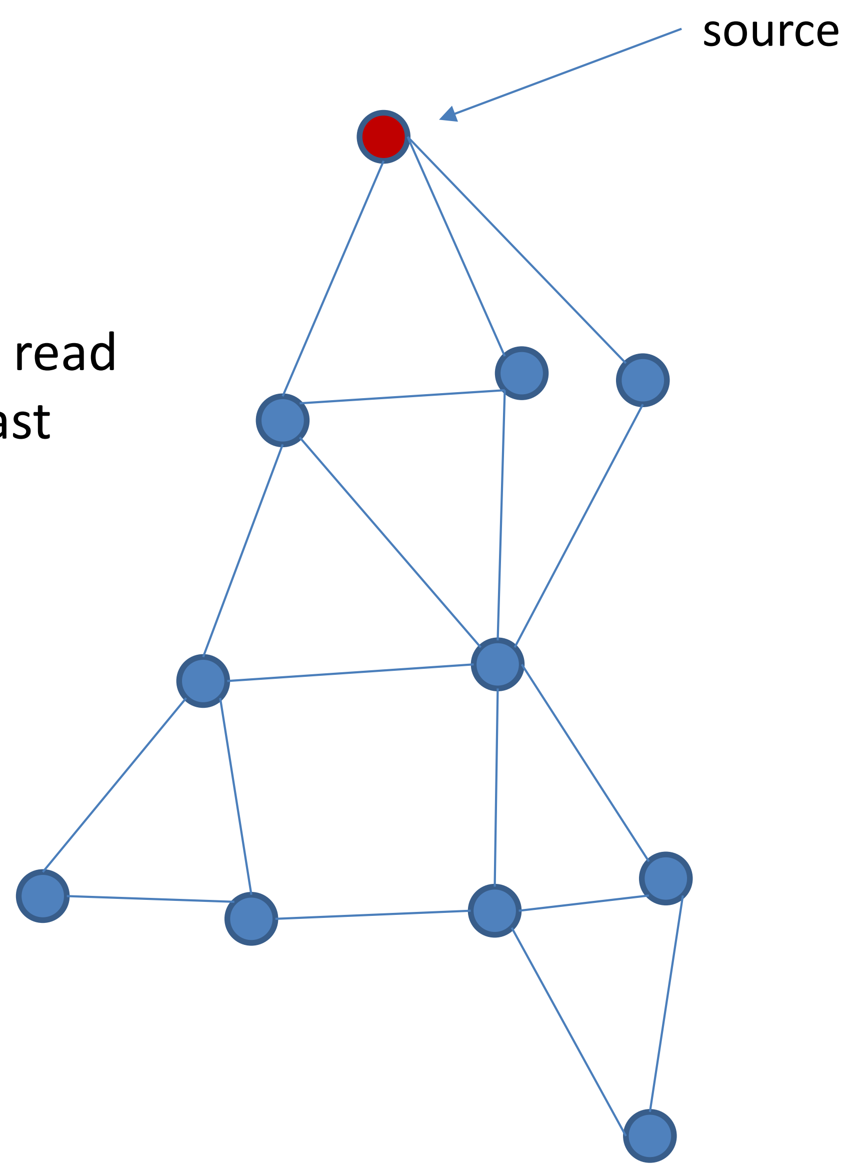


Problem: Breadth First Search

Can we do better?

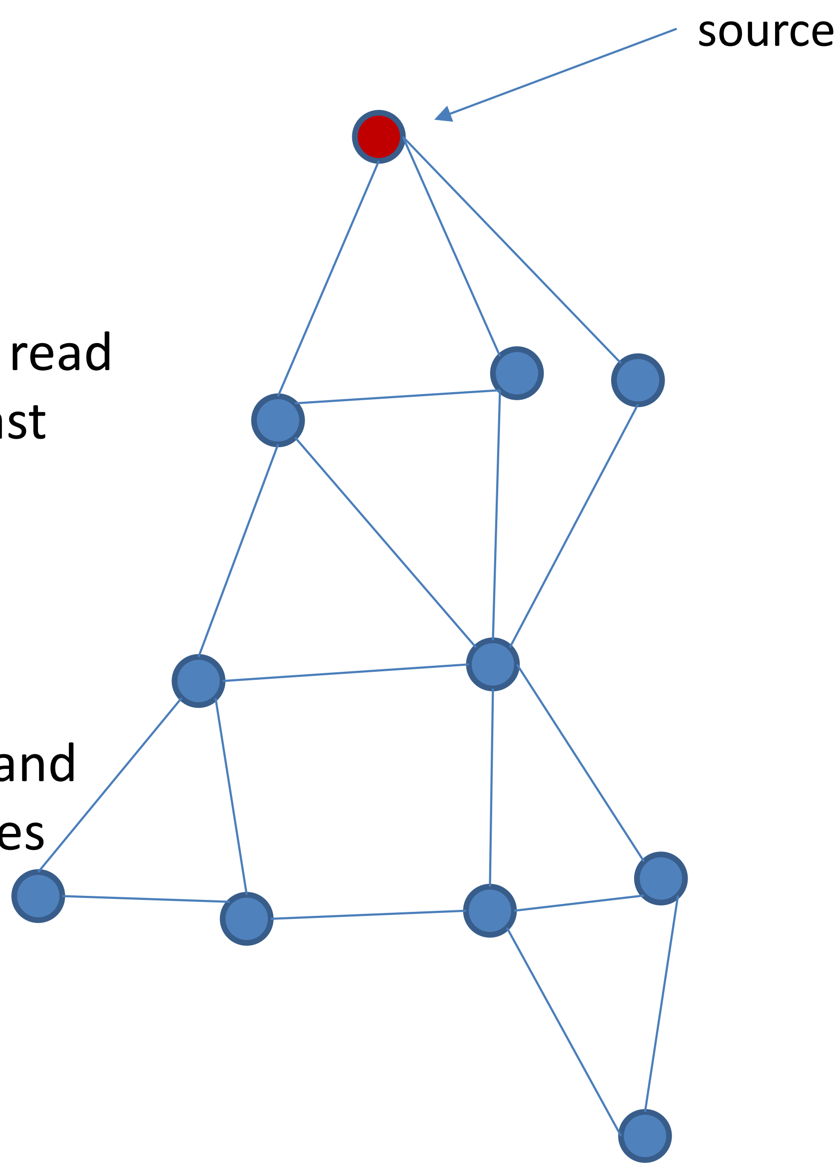
Unlikely in dense graph.

- 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.

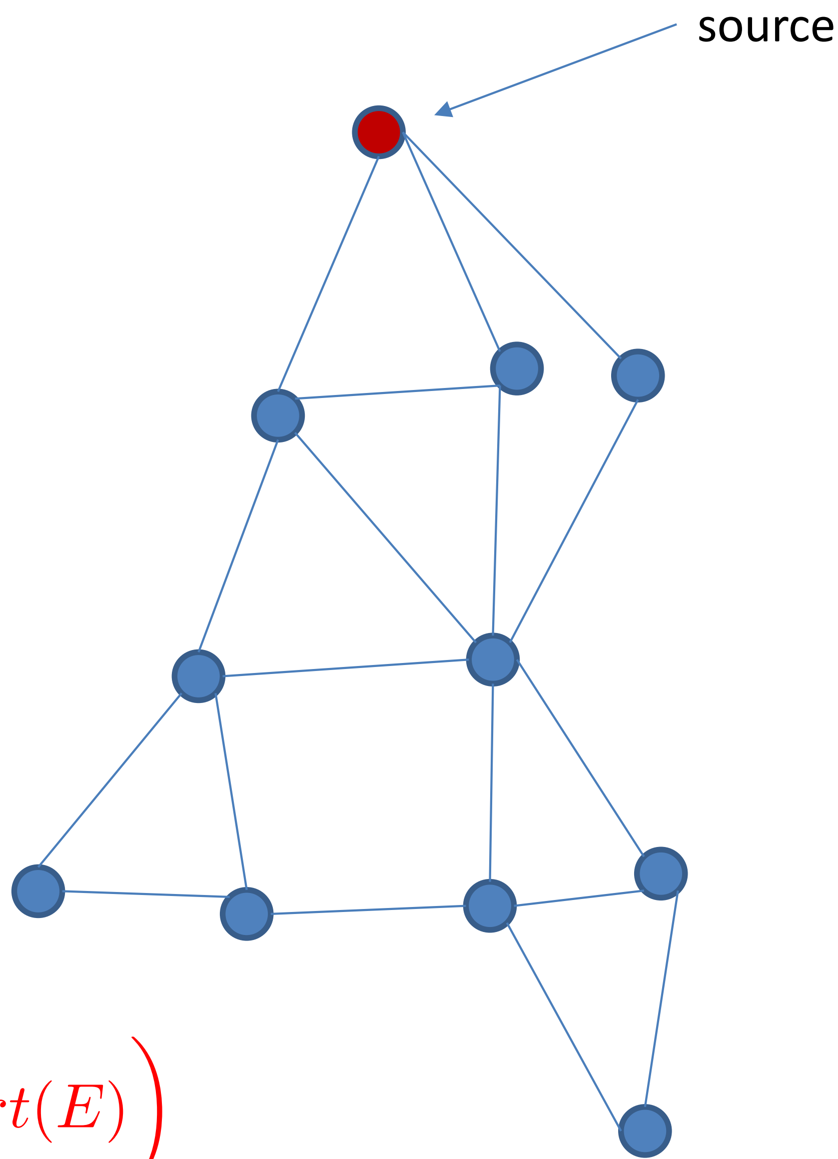
Problem: Breadth First Search

Can we do better?

Sparse graph

Store all edges in one array.

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



If |E| = O(|V|) then: $O\left(\frac{|V|}{B} + sort(E)\right)$

Summary

Today: Graph Algorithms

Breadth-First-Search

- *Sorting your graph*

MIS

- *Luby's Algorithm*
- *Cache-efficient implementation*

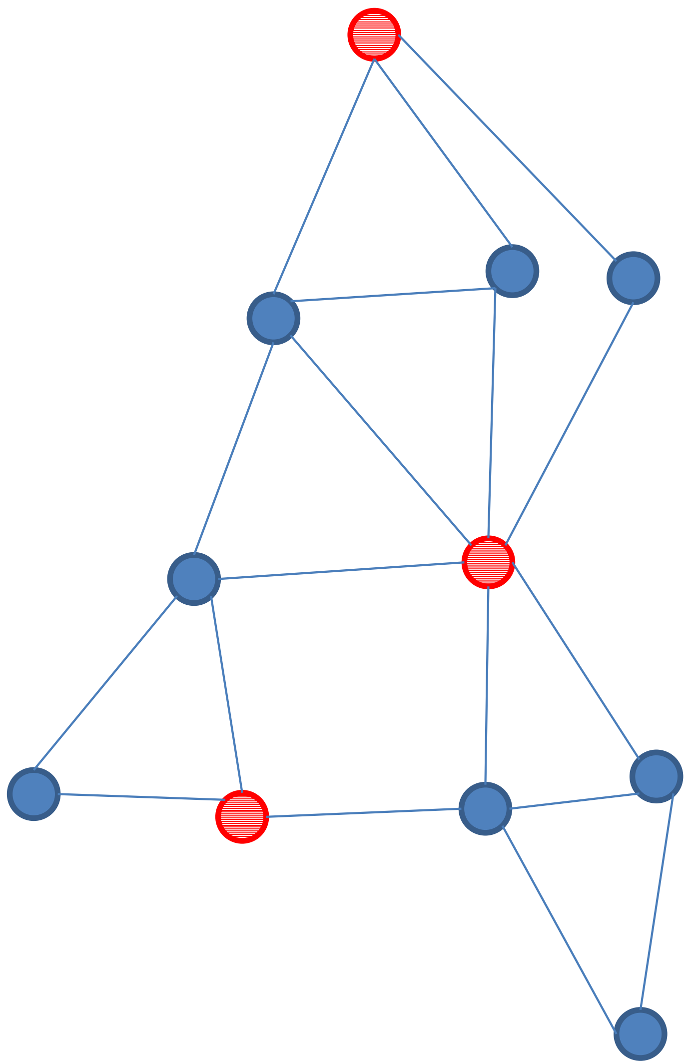
MST

- *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

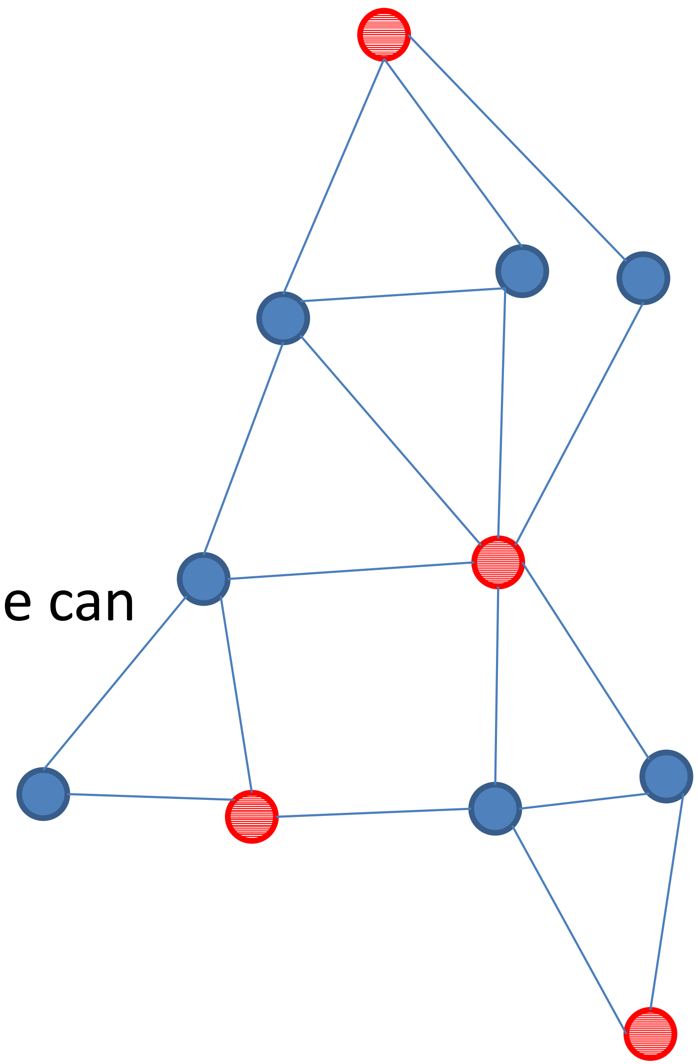
Independent Set:

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:

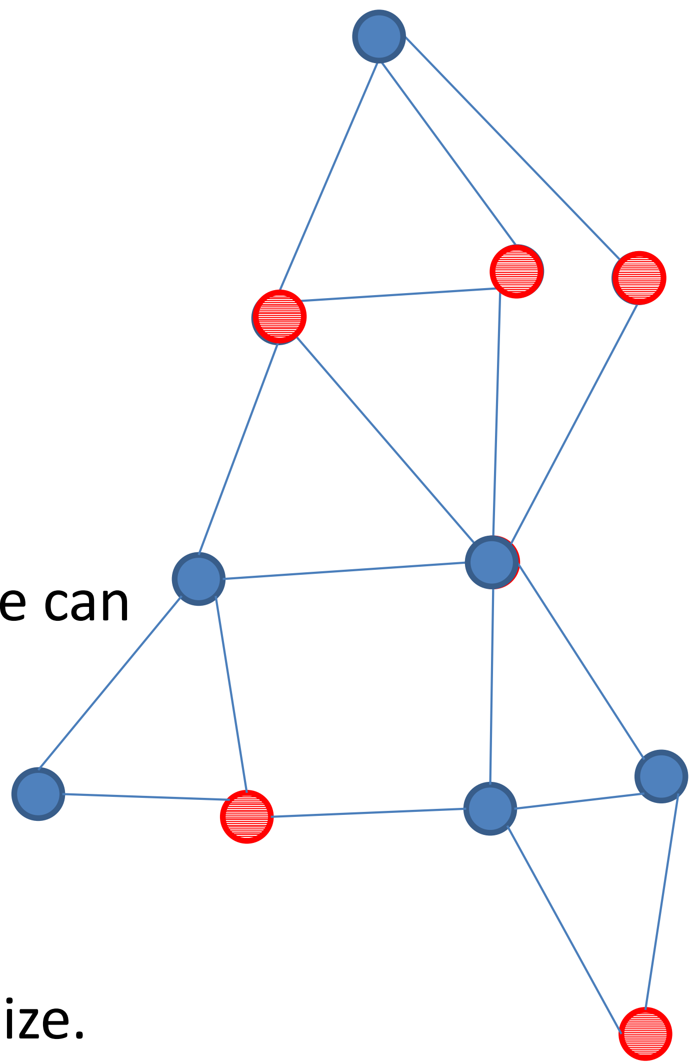
A set of nodes S so that no two neighbors are in S.

Maximal Independent Set:

An independent set S where no node can be added.

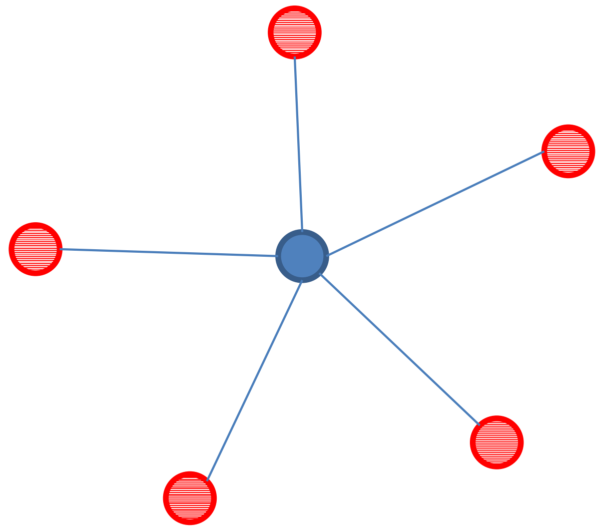
Maximum Independent Set:

An independent set S of maximum size.



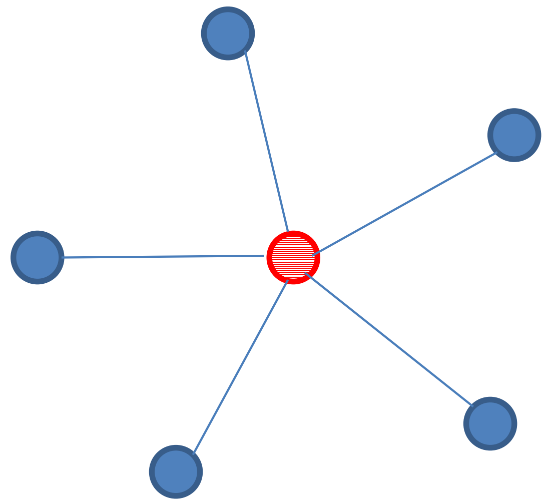
Maximal Independent Set

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.

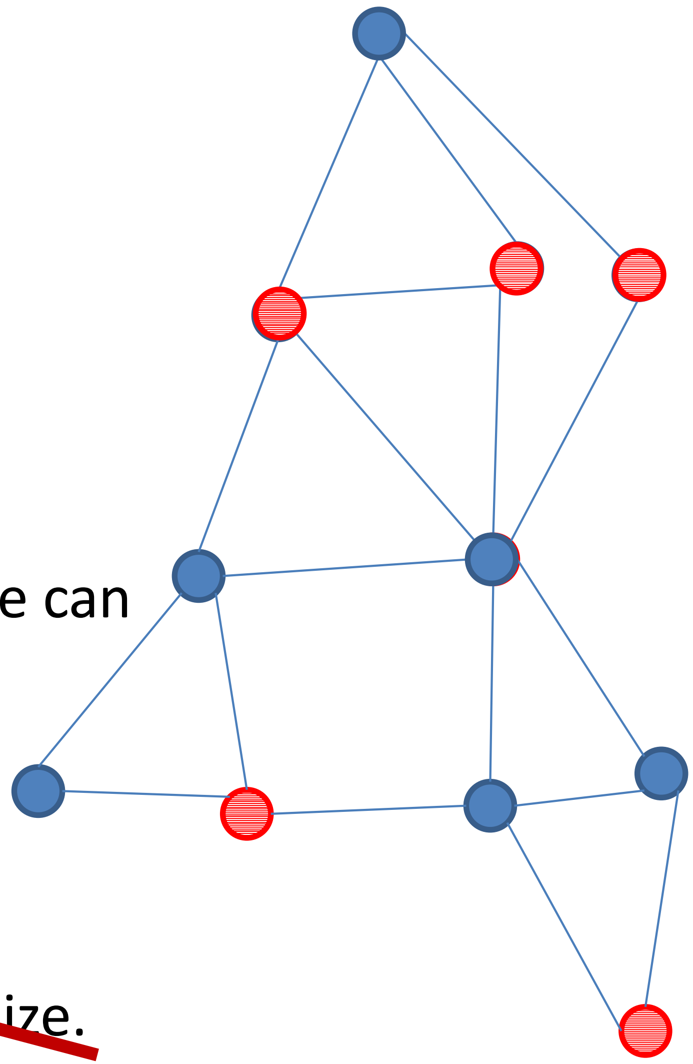
Maximal Independent Set

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:~~

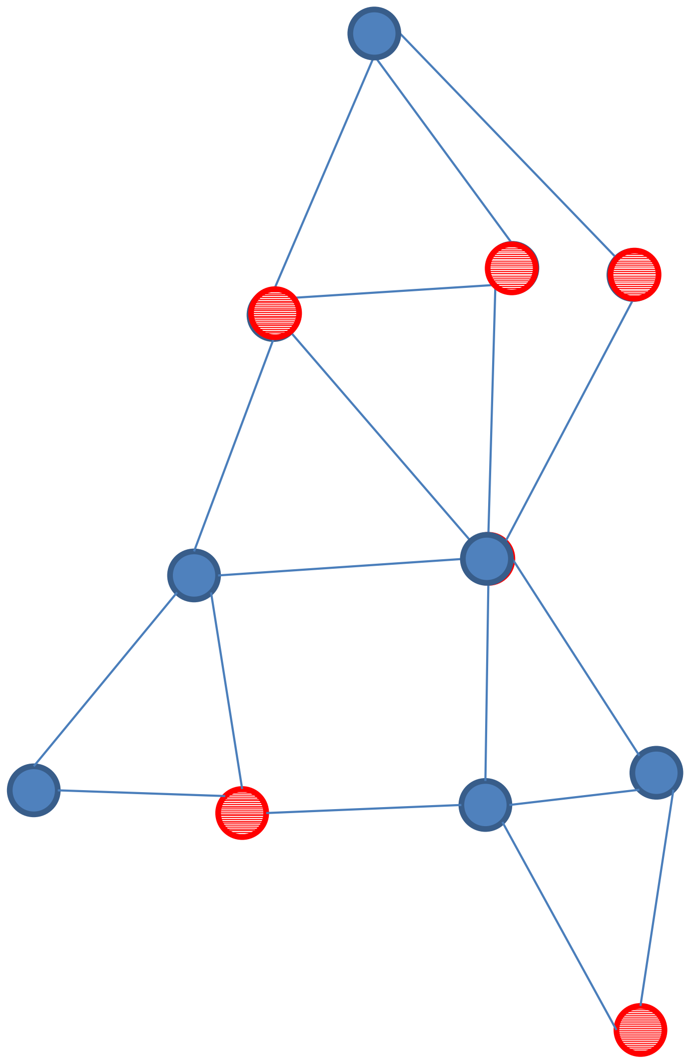
NP-Hard

~~An independent set S of maximum size.~~

Maximal Independent Set

Greedy MIS Algorithm:

- S = empty set
- for every node v:
  - If no neighbor of v is in S, then add v to S.



Maximal Independent Set

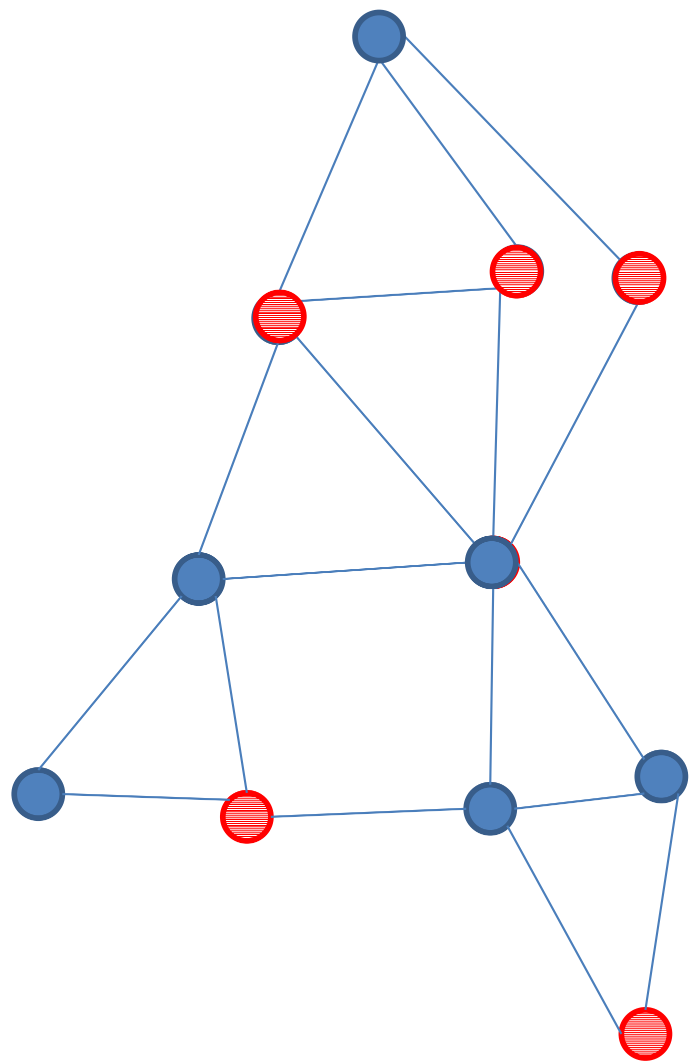
Greedy MIS Algorithm:

- S = 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:

degree of node u

- $S = \varnothing$
- 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:

degree of node u

- $S = \varnothing$
- 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]

Luby's Algorithm

Claim 1:

The set S is a maximal independent set.

Luby's Algorithm

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

Luby's Algorithm

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 → 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 $1/2d(u)$.
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

Analysis

Define: E_j = edges at start of iteration j.

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

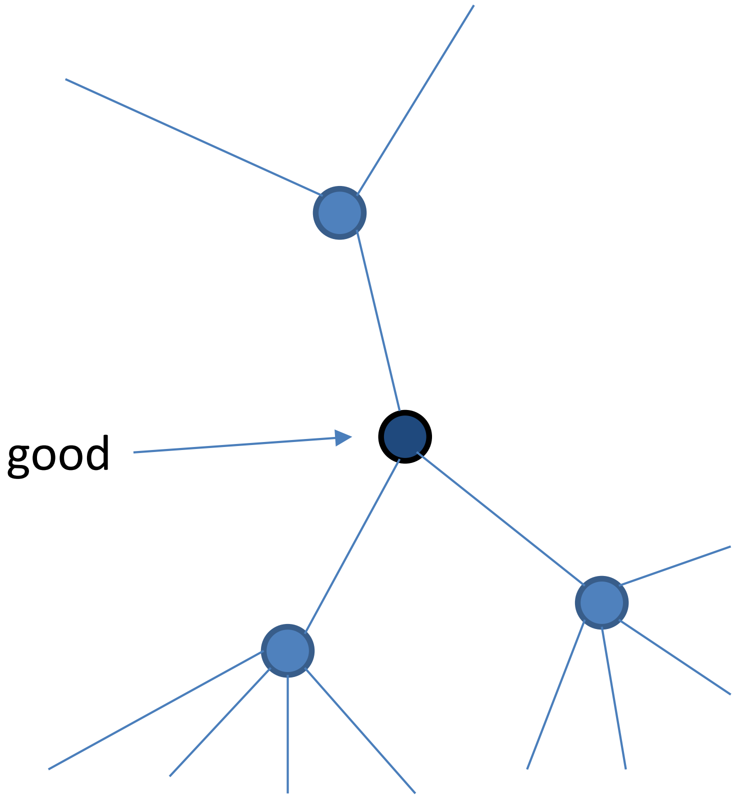
$$\mathbf{E}[E_j \mid E_{j-1}] \leq \alpha E_{j-1}$$

Idea: reduce the number of edges by a constant fraction in each iteration.

Luby's Algorithm

Analysis

Define: node w is good if ≥ 1/3 neighbors have smaller degree than w.



Luby's Algorithm

Analysis

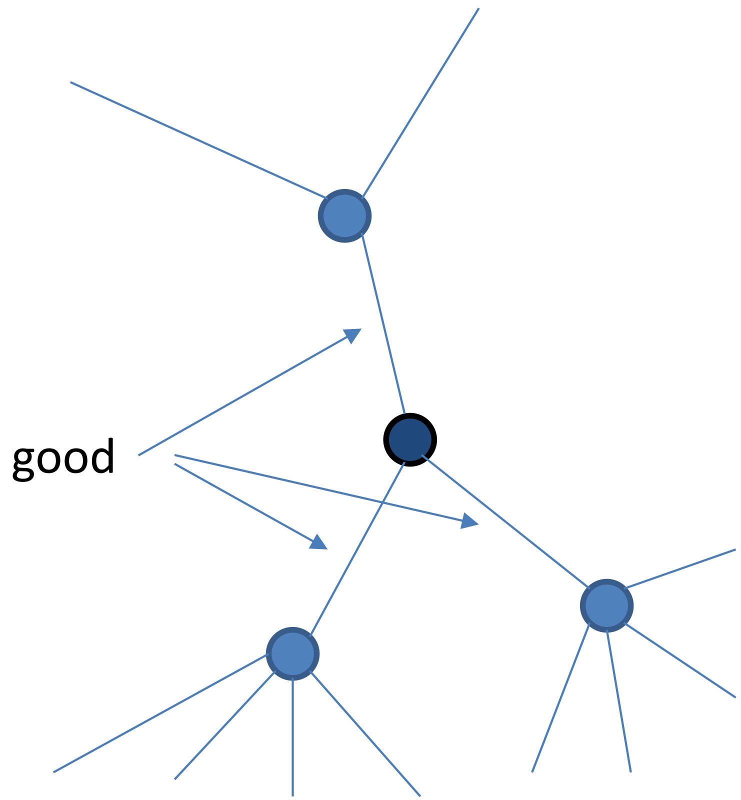
Define: node w is <u>good</u> if ≥ 1/3 neighbors have smaller degree than w.

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.



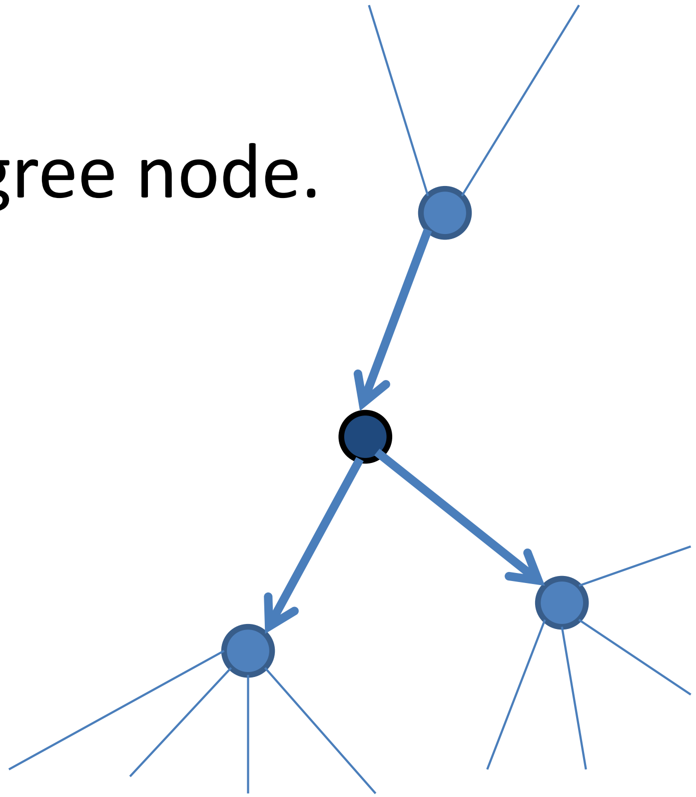
Luby's Algorithm

Analysis

Claim: At least half of all edges are good.

Proof:

Orient each edge TO the higher degree node.



Luby’s Algorithm

Analysis

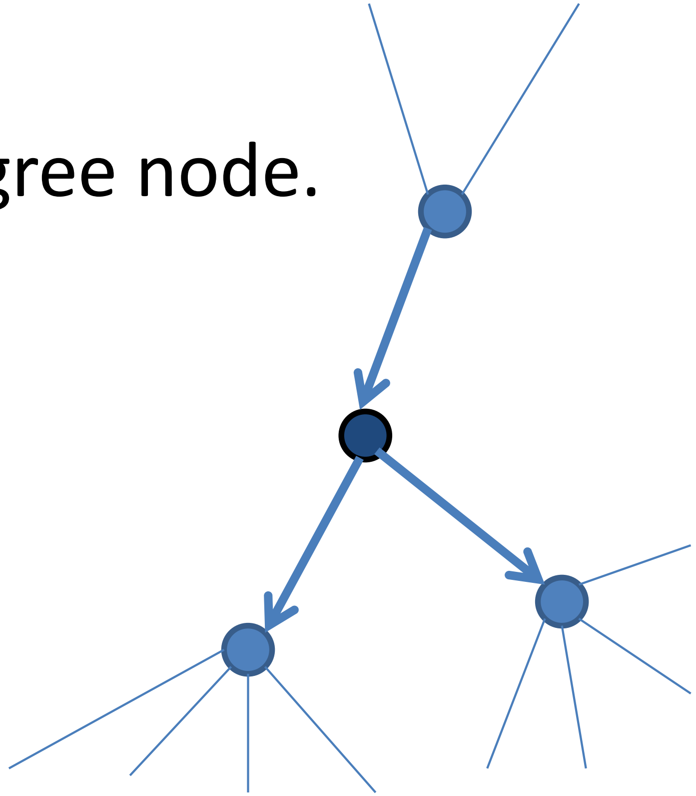
Claim: At least half of all edges are good.

Proof:

Orient each edge TO the higher degree node.

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

.



good ➔ ≥ 1/3 have smaller degree

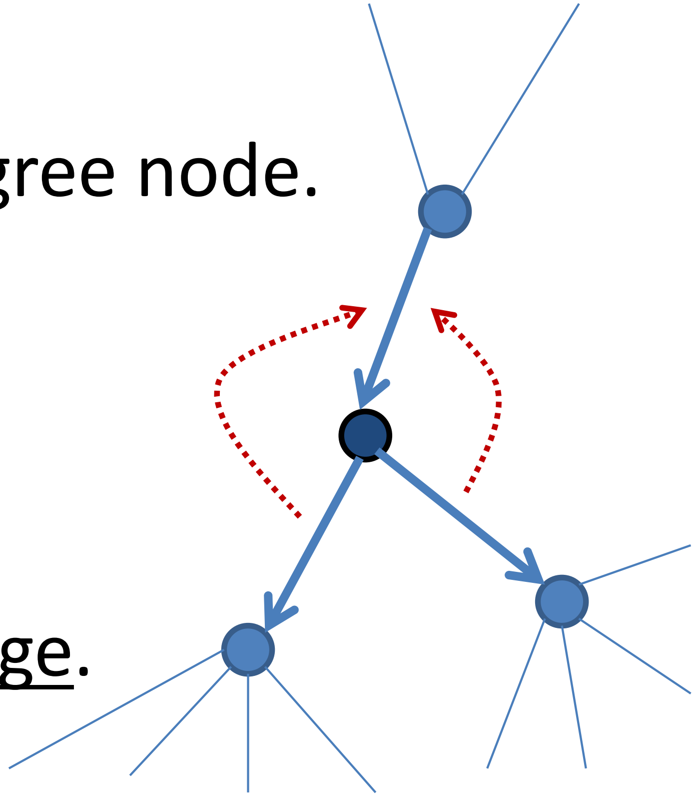
Luby's Algorithm

Analysis

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…)

Luby's Algorithm

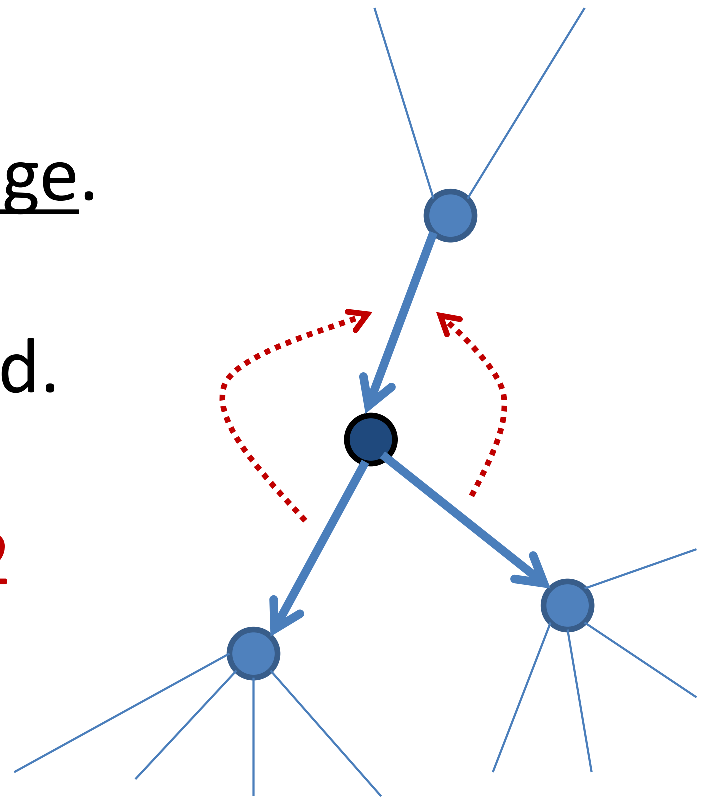
Analysis

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.



Luby's Algorithm

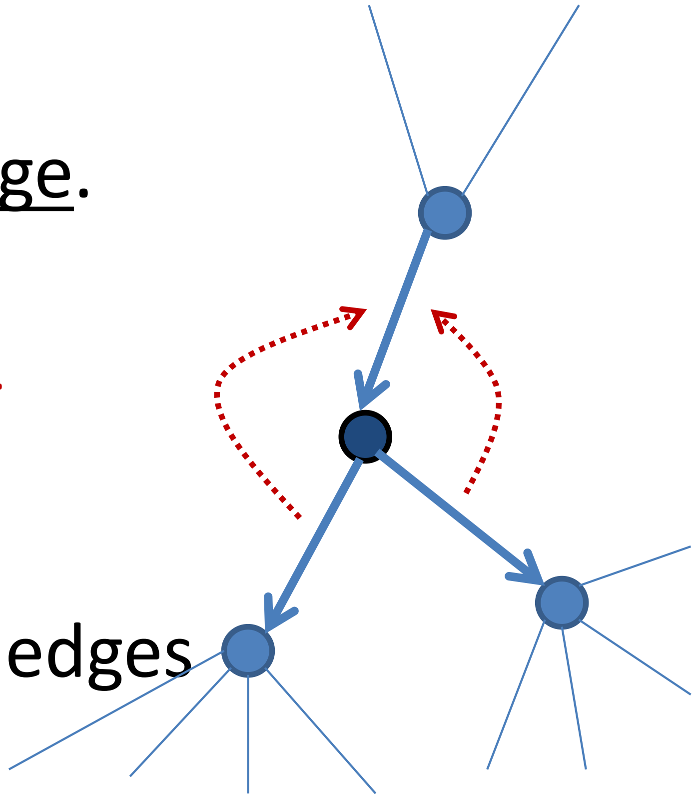
Analysis

Claim: At least half of all edges are good.

Proof:

Assign <u>two OUT edges</u> to <u>one IN edge</u>.

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



If there are B bad nodes, then ≥ 2B edges total in graph.

Luby's Algorithm

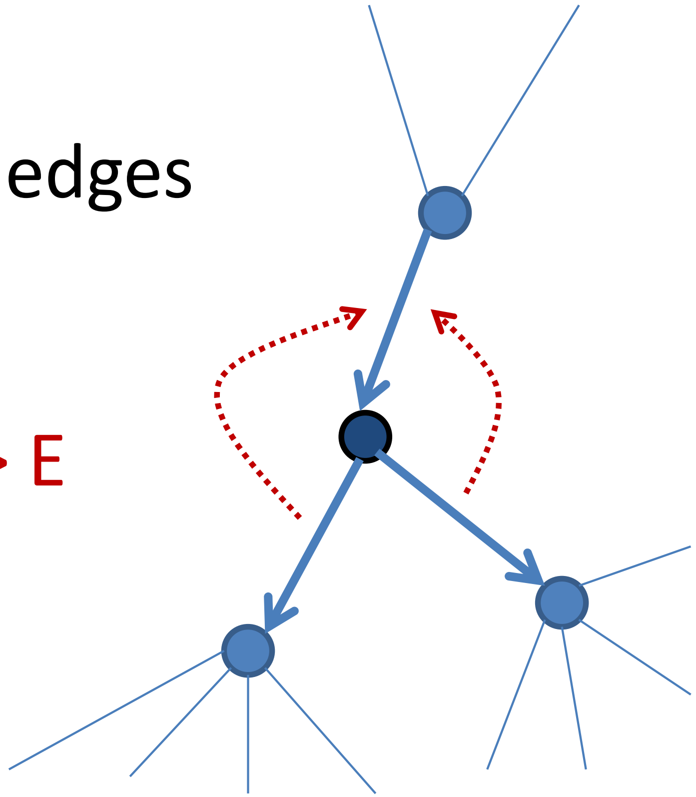
Analysis

Claim: At least half of all edges are good.

Proof:
If there are B bad nodes, then ≥ 2B edges total in graph.

If there are > E/2 bad nodes, then > E edges total in graph ➔ impossible.

➔ > E/2 good nodes.

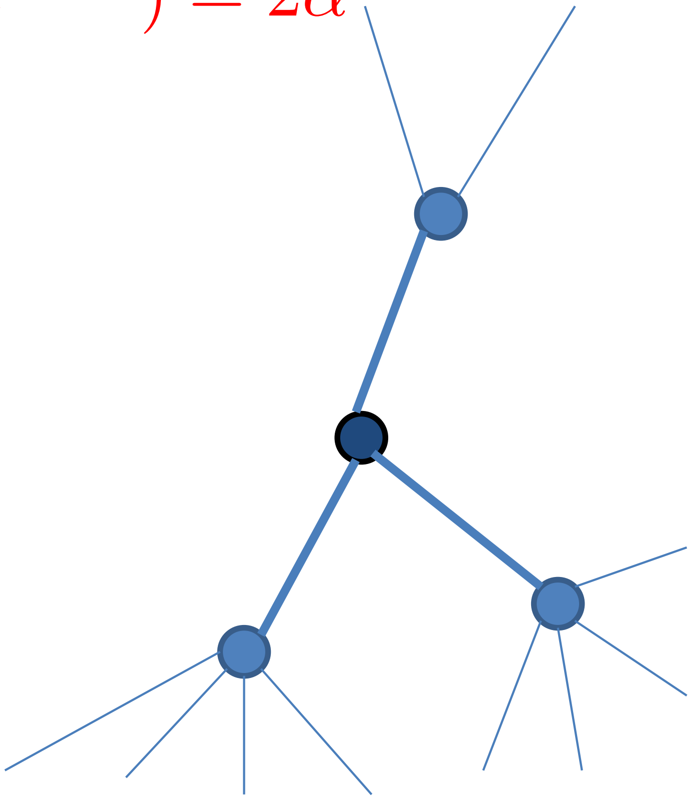


Luby's Algorithm

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:

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

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

Show at least one neighbor of v with smaller degree is marked!

Luby's Algorithm

Claim: If v is good, then:

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

$$\begin{aligned}
\Pr\left[\text{no nbr of } v \text{ marked}\right] &\leq \Pr\left[\text{no nbr of } v \text{ with smaller degree marked}\right] \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}]
\end{aligned}$$

Nodes are marked independently.

Luby's Algorithm

Claim: If v is good, then:

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

$$\begin{aligned}
\Pr\left[\text{no nbr of } v \text{ marked}\right] &\leq \Pr\left[\text{no nbr of } v \text{ with smaller degree marked}\right] \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}] \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right)
\end{aligned}$$

The probability that a node w is marked is 1/2d(w).

Luby's Algorithm

Claim: If v is good, then:

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

$$\begin{aligned}
\Pr\left[\text{no nbr of } v \text{ marked}\right] &\leq \Pr\left[\text{no nbr of } v \text{ with smaller degree marked}\right] \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \Pr[w \text{ not marked}] \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(w)}\right) \\
&\leq \prod_{w \text{ smaller degree nbr of } v} \left(1 - \frac{1}{2d(v)}\right)
\end{aligned}$$

By assumption, d(w) < d(v).

Luby's Algorithm

Claim: If v is good, then:

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

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

At least d(v)/3 neighbors with smaller degree because v is good.

Luby's Algorithm

Claim: If v is good, then:

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

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

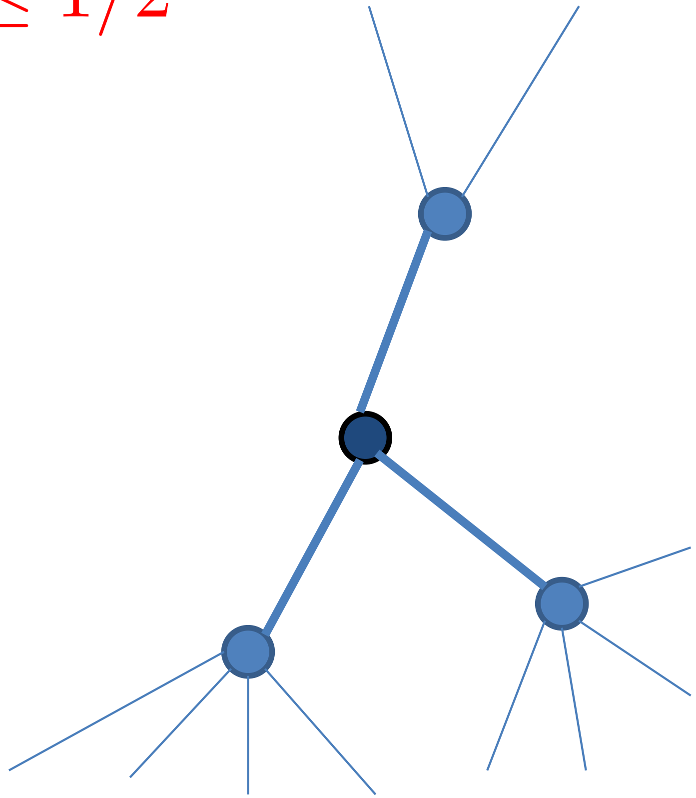
$\boxed{(1-1/x)^x \leq e^{-1}}$

Luby's Algorithm

Analysis

Claim: If w is marked, then:

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



Luby's Algorithm

Claim: If w is marked, then:

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

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

Only unmark if higher degree neighbor is marked.

Luby's Algorithm

Claim: If w is marked, then:

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

$$\begin{aligned}\Pr\left[\text{unmark } w \mid w \text{ marked}\right] &\leq \Pr[\text{higher degree neighbor of } w \text{ marked}] \\ &\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)}\end{aligned}$$

Union bound...

Luby's Algorithm

Claim: If w is marked, then:

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

$$\begin{aligned}
\Pr\left[\text{unmark } w \mid w \text{ marked}\right] &\leq \Pr[\text{higher degree neighbor of } w \text{ marked}] \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)} \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)}
\end{aligned}$$

By assumption, d(w) < d(z).

Luby's Algorithm

Claim: If w is marked, then:

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

$$\begin{aligned}
\Pr\left[\text{unmark } w \mid w \text{ marked}\right] &\leq \Pr[\text{higher degree neighbor of } w \text{ marked}] \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)} \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)} \\
&\leq \frac{d(w)}{2d(w)}
\end{aligned}$$

Node w has d(w) neighbors.

Luby's Algorithm

Claim: If w is marked, then:

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

$$\begin{aligned}
\Pr\left[\text{unmark } w \mid w \text{ marked}\right] &\leq \Pr[\text{higher degree neighbor of } w \text{ marked}] \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(z)} \\
&\leq \sum_{z \text{ higher degree neighbor of } w} \frac{1}{2d(w)} \\
&\leq \frac{d(w)}{2d(w)} \\
&\leq \frac{1}{2}
\end{aligned}$$

Luby's Algorithm

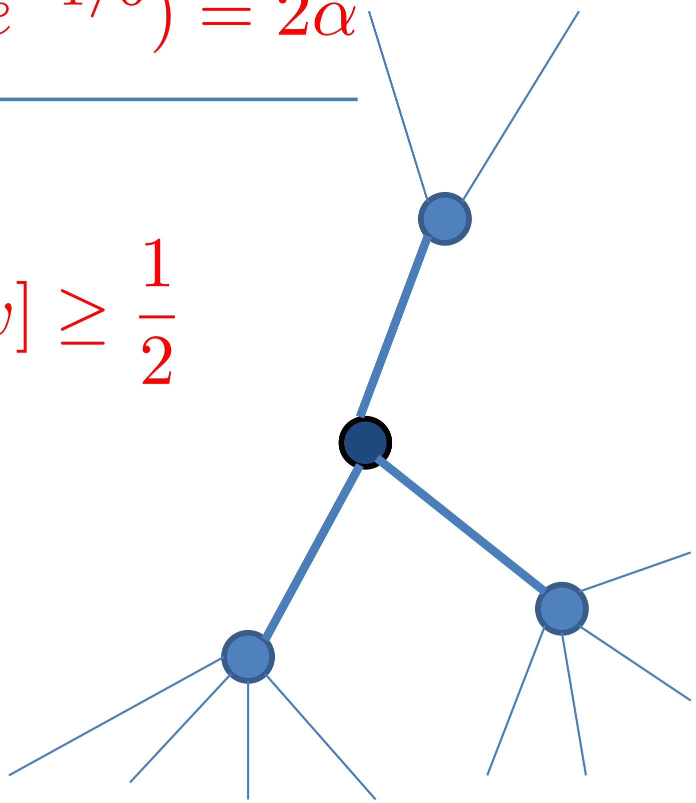
Analysis

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:

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

Claim: If w is marked, then:

$$\Pr[\text{stay marked } w \mid \text{marked } w] \geq \frac{1}{2}$$

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:

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

Claim: If w is marked, then:

$$\Pr[\text{stay marked } w \mid \text{marked } w] \geq \frac{1}{2}$$

Claim: If v is good, then:

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

Luby's Algorithm

Analysis

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:

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

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

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

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

Luby's Algorithm

Analysis

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

$$\mathbf{E}[E_j] \quad = \quad \mathbf{E}[\mathbf{E}[E_j | E_{j-1}]]$$

Law of Total Expectation

Luby's Algorithm

Analysis

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

$$\begin{aligned} \mathbf{E}[E_j] &= \mathbf{E}[\mathbf{E}[E_j | E_{j-1}]] \\ &\leq \mathbf{E}[E_{j-1}](1 - \alpha/2) \end{aligned}$$

Substitution.

Luby's Algorithm

Analysis

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

$$\begin{aligned}
\mathbf{E}[E_j] &= \mathbf{E}[\mathbf{E}[E_j | E_{j-1}]] \\
&\leq \mathbf{E}[E_{j-1}](1 - \alpha/2) \\
&\leq |E|(1 - \alpha/2)^j
\end{aligned}$$

Induction.
Note that $E_0 = |E|$.

Luby's Algorithm

Analysis

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

$$\begin{aligned} \mathbf{E}[E_j] &= \mathbf{E}[\mathbf{E}[E_j | E_{j-1}]] \\ &\leq \mathbf{E}[E_{j-1}](1 - \alpha/2) \\ &\leq |E|(1 - \alpha/2)^j \end{aligned}$$

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

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

Luby's Algorithm

Analysis

Theorem:

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

Luby's Algorithm

Expected time?

Maximal Independent Set

Luby's Algorithm:

- $S = \varnothing$
- 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.

Luby's Algorithm

Expected time?

$$O(E + (1 - \alpha/2)E + (1 - \alpha/2)^2 E + (1 - \alpha/2)^3 E + \ldots) = O(E)$$

Luby's Algorithm

Analysis

Theorem:

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

Cache Efficient??

Luby's Algorithm:

- $S = \varnothing$
- 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.

Cache-Efficient Luby's

Setup

Initially:

Assume that all the edges are in a single array.

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

Ex:

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

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

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:

concatenated adjacency lists with extra bits

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)]

Cache-Efficient Luby's

Setup

Initially:

concatenated adjacency lists with extra bits

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).

<u>Cache-efficient:</u>

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.

<u>Cache-efficient:</u>

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:

(a,b)	**(a,d)**	**(a,e)**	**(b,a)**	**(b,c)**	**(c,b)**	**(d,a)**	**(d,e)**	**(e,a)**	**(e,d)**
3	**3**	**3**	**2**	**2**	**1**	**2**	**2**	**2**	**2**
X	**X**	**X**			**X**			**X**	**X**

Sort by second:

(b,a)	**(d,a)**	**(e,a)**	**(a,b)**	**(c,b)**	**(b,c)**	**(a,d)**	**(e,d)**	**(a,e)**	**(d,e)**
2	**2**	**2**	**3**	**1**	**2**	**3**	**2**	**3**	**2**
		X	**X**	**X**		**X**	**X**	**X**	

Cache Efficient Luby's

Sort by first:

(a,b)	**(a,d)**	**(a,e)**	**(b,a)**	**(b,c)**	**(c,b)**	**(d,a)**	**(d,e)**	**(e,a)**	**(e,d)**
3	**3**	**3**	**2**	**2**	**1**	**2**	**2**	**2**	**2**
X	**X**	**X**			**X**			**X**	**X**

Sort by second:

(b,a)	**(d,a)**	**(e,a)**	**(a,b)**	**(c,b)**	**(b,c)**	**(a,d)**	**(e,d)**	**(a,e)**	**(d,e)**
2	**2**	**2**	**3**	**1**	**2**	**3**	**2**	**3**	**2**
		X	**X**	**X**		**X**	**X**	**X**	

Scan neighbors of node a.
Do not unmark a.

Cache Efficient Luby's

Sort by first:

(a,b)	(a,d)	(a,e)	(b,a)	(b,c)	(c,b)	(d,a)	(d,e)	(e,a)	(e,d)
3	3	3	2	2	1	2	2	2	2
X	X	X			X			X	X

Sort by second:

(b,a)	(d,a)	(e,a)	(a,b)	(c,b)	(b,c)	(a,d)	(e,d)	(a,e)	(d,e)
2	2	2	3	1	2	3	2	3	2
		X	X	X		X	X	X	

Scan neighbors of node b.

If b were marked, unmark b because a is marked.

Cache Efficient Luby's

Sort by first:

(a,b)	(a,d)	(a,e)	(b,a)	(b,c)	(c,b)	(d,a)	(d,e)	(e,a)	(e,d)
3	3	3	2	2	1	2	2	2	2
X	X	X			X			X	X

Sort by second:

(b,a)	(d,a)	(e,a)	(a,b)	(c,b)	(b,c)	(a,d)	(e,d)	(a,e)	(d,e)
2	2	2	3	1	2	3	2	3	2
		X	X	X		X	X	X	

Scan neighbors of node c.
None are marked.

Cache Efficient Luby's

Sort by first:

(a,b)	(a,d)	(a,e)	(b,a)	(b,c)	(c,b)	(d,a)	(d,e)	(e,a)	(e,d)
3	3	3	2	2	1	2	2	2	2
X	X	X			X			X	X

Sort by second:

(b,a)	(d,a)	(e,a)	(a,b)	(c,b)	(b,c)	(a,d)	(e,d)	(a,e)	(d,e)
2	2	2	3	1	2	3	2	3	2
		X	X	X		X	X	X	

Scan neighbors of node d.

Cache Efficient Luby's

Sort by first:

(a,b)	(a,d)	(a,e)	(b,a)	(b,c)	(c,b)	(d,a)	(d,e)	(e,a)	(e,d)
3	3	3	2	2	1	2	2	2	2
X	X	X			X			X	X

Sort by second:

(b,a)	(d,a)	(e,a)	(a,b)	(c,b)	(b,c)	(a,d)	(e,d)	(a,e)	(d,e)
2	2	2	3	1	2	3	2	3	2
		X	X	X		X	X	X	

Scan neighbors of node e.
Unmark e because a is marked and has higher degree.

Cache Efficient Luby's

Sort by first:

(a,b)	(a,d)	(a,e)	(b,a)	(b,c)	(c,b)	(d,a)	(d,e)	(e,a)	(e,d)
3	3	3	2	2	1	2	2	2	2
X	X	X			X				

Sort by second:

(b,a)	(d,a)	(e,a)	(a,b)	(c,b)	(b,c)	(a,d)	(e,d)	(a,e)	(d,e)
2	2	2	3	1	2	3	2	3	2
		X	X	X		X	X	X	

Scan neighbors of node e.
Unmark e because a is marked and has higher degree.

Cache Efficient Luby's

Sort by first:

(a,b)	**(a,d)**	**(a,e)**	**(b,a)**	**(b,c)**	**(c,b)**	**(d,a)**	**(d,e)**	**(e,a)**	**(e,d)**
3	**3**	**3**	**2**	**2**	**1**	**2**	**2**	**2**	**2**
X	**X**	**X**			**X**				

Sort by second:

(b,a)	**(d,a)**	**(e,a)**	**(a,b)**	**(c,b)**	**(b,c)**	**(a,d)**	**(e,d)**	**(a,e)**	**(d,e)**
2	**2**	**2**	**3**	**1**	**2**	**3**	**2**	**3**	**2**
		X	**X**	**X**		**X**	**X**	**X**	

$$O(sort(E) + E/B)$$

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.

<u>Cache-efficient:</u>

Make a copy E'.

Sort by 2nd component of edge (., u).

Iterate and unmark if higher degree neighbor is marked.

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.

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)

(b,a) 2	(c,a) 2	(e,a) 2	(b,d) 2	(h,d) 1	(d,f) 2	(c,f) 1	(d,h) 2		

S (sorted by first)

(a,b) 3 X	(a,c) 3 X	(a,e) 3 X	(f,d) 2 X	(f,c) 2 X					

Scan neighbors of node a.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D							

S (sorted by first)

(a,b)	(a,c)	(a,e)	(f,d)	(f,c)					
3	3	3	2	2					
X	X	X	X	X					

Scan neighbors of node a.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D							

S (sorted by first)

(a,b)	(a,c)	(a,e)	(f,d)	(f,c)					
3	3	3	2	2					
X	X	X	X	X					

Scan neighbors of node d.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D							

S (sorted by first)

(a,b)	(a,c)	(a,e)	(f,d)	(f,c)					
3	3	3	2	2					
X	X	X	X	X					

Scan neighbors of node f.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D			D	D			

S (sorted by first)

(a,b)	(a,c)	(a,e)	(f,d)	(f,c)					
3	3	3	2	2					
X	X	X	X	X					

Scan neighbors of node f.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a) 2 D	(c,a) 2 D	(e,a) 2 D	(b,d) 2	(h,d) 1	(d,f) 2 D	(c,f) 1 D	(d,h) 2		

S (sorted by first)

(a,b) 3 X	(a,c) 3 X	(a,e) 3 X	(f,d) 2 X	(f,c) 2 X					

Scan neighbors of node h.
Mark to delete if neighbor is marked.

Cache Efficient Luby's

E (sorted by second)

(b,a) **2** **D**	**(c,a)** **2** **D**	**(e,a)** **2** **D**	**(b,d)** **2**	**(h,d)** **1**	**(d,f)** **2** **D**	**(c,f)** **1** **D**	**(d,h)** **2**		

Cache Efficient Luby's

E (sorted by first)

(b,a)	**(b,d)**	**(c,a)**	**(c,f)**	**(d,f)**	**(d,h)**	**(e,a)**	**(h,d)**		
2	**2**	**2**	**1**	**2**	**2**	**2**	**1**		
D	**D**	**D**	**D**	**D**	**D**	**D**			

Sort and mark all associated with same node as deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a) **2** **D**	**(b,d)** **2** **D**	**(c,a)** **2** **D**	**(c,f)** **1** **D**	**(d,f)** **2** **D**	**(d,h)** **2** **D**	**(e,a)** **2** **D**	**(h,d)** **1**		

E (sorted by second)

(b,a) **2** **D**	**(c,a)** **2** **D**	**(e,a)** **2** **D**	**(b,d)** **2** **D**	**(h,d)** **1**	**(d,f)** **2** **D**	**(c,f)** **1** **D**	**(d,h)** **2** **D**		

Copy and sort.

Cache Efficient Luby's

E (sorted by first)

(b,a) 2 D	**(b,d) 2 D**	**(c,a) 2 D**	**(c,f) 1 D**	**(d,f) 2 D**	**(d,h) 2 D**	**(e,a) 2 D**	**(h,d) 1**		

E (sorted by second)

(b,a) 2 D	**(c,a) 2 D**	**(e,a) 2 D**	**(b,d) 2 D**	**(h,d) 1**	**(d,f) 2 D**	**(c,f) 1 D**	**(d,h) 2 D**		

Scan and mark deleted if any neighbor is marked deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a)	(b,d)	(c,a)	(c,f)	(d,f)	(d,h)	(e,a)	(h,d)		
2	2	2	1	2	2	2	1		
D	D	D	D	D	D	D			

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D	D		D	D	D		

Scan and mark deleted if any neighbor is marked deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a)	**(b,d)**	**(c,a)**	**(c,f)**	**(d,f)**	**(d,h)**	**(e,a)**	**(h,d)**		
2	**2**	**2**	**1**	**2**	**2**	**2**	**1**		
D	**D**	**D**	**D**	**D**	**D**	**D**			

E (sorted by second)

(b,a)	**(c,a)**	**(e,a)**	**(b,d)**	**(h,d)**	**(d,f)**	**(c,f)**	**(d,h)**		
2	**2**	**2**	**2**	**1**	**2**	**1**	**2**		
D	**D**	**D**	**D**		**D**	**D**	**D**		

Scan and mark deleted if any neighbor is marked deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a)	(b,d)	(c,a)	(c,f)	(d,f)	(d,h)	(e,a)	(h,d)		
2	2	2	1	2	2	2	1		
D	D	D	D	D	D	D			

E (sorted by second)

(b,a)	(c,a)	(e,a)	(b,d)	(h,d)	(d,f)	(c,f)	(d,h)		
2	2	2	2	1	2	1	2		
D	D	D	D		D	D	D		

Scan and mark deleted if any neighbor is marked deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a)	**(b,d)**	**(c,a)**	**(c,f)**	**(d,f)**	**(d,h)**	**(e,a)**	**(h,d)**		
2	**2**	**2**	**1**	**2**	**2**	**2**	**1**		
D	**D**	**D**	**D**	**D**	**D**	**D**	**D**		

E (sorted by second)

(b,a)	**(c,a)**	**(e,a)**	**(b,d)**	**(h,d)**	**(d,f)**	**(c,f)**	**(d,h)**		
2	**2**	**2**	**2**	**1**	**2**	**1**	**2**		
D	**D**	**D**	**D**		**D**	**D**	**D**		

Scan and mark deleted if any neighbor is marked deleted.

Cache Efficient Luby's

E (sorted by first)

(b,a) 2 D	**(b,d) 2 D**	**(c,a) 2 D**	**(c,f) 1 D**	**(d,f) 2 D**	**(d,h) 2 D**	**(e,a) 2 D**	**(h,d) 1 D**		

new array

Copy anything left to a new array E for the next iteration.

Cache Efficient Luby's

E (sorted by first)

(b,a) 2 D	**(b,d) 2 D**	**(c,a) 2 D**	**(c,f) 1 D**	**(d,f) 2 D**	**(d,h) 2 D**	**(e,a) 2 D**	**(h,d) 1 D**		

new array

$$O(sort(E) + E/B)$$

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.
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 + sort(E)) time, in expectation.

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

Summary

Today: Graph Algorithms

Breadth-First-Search

- *Sorting your graph*

MIS

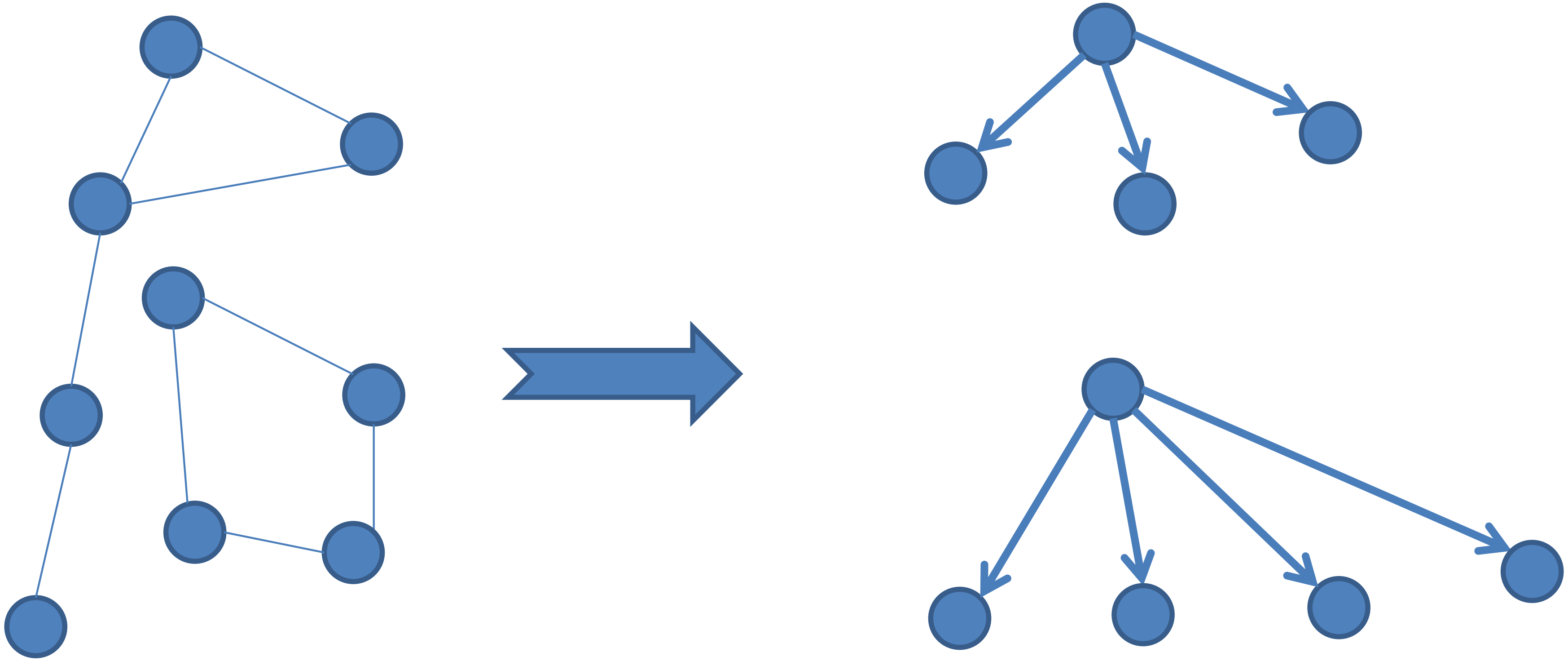
- *Luby's Algorithm*
- *Cache-efficient implementation*

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: E1 and E2.

Cache-Efficient Connectivity

Algorithm Idea

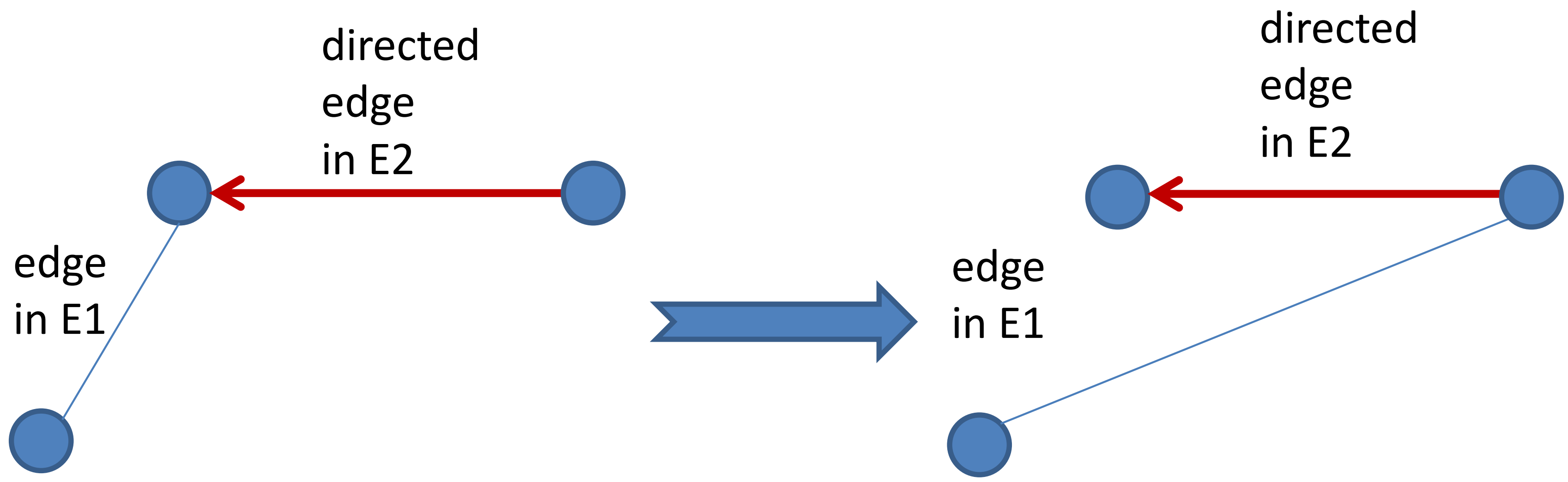
1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 → depth 1 trees.

Base case:
One edge → done.

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.



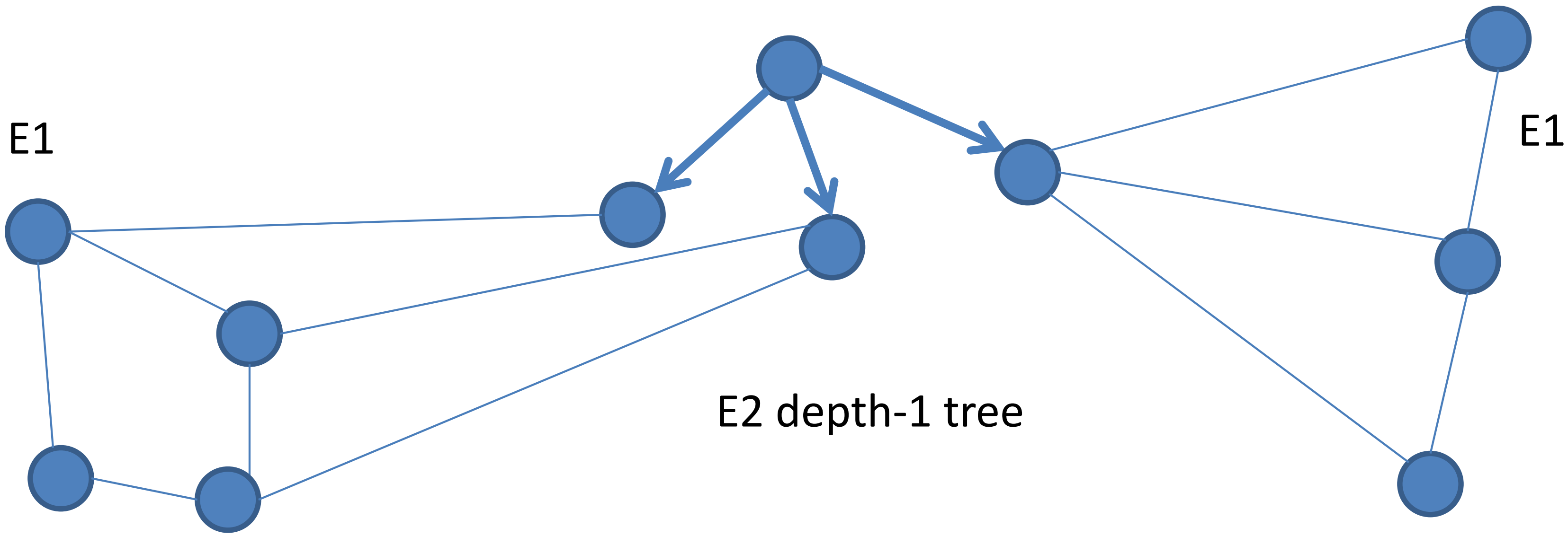
Only “root” nodes in E2 are connected to E1.

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.

Claim: does not change connected components.

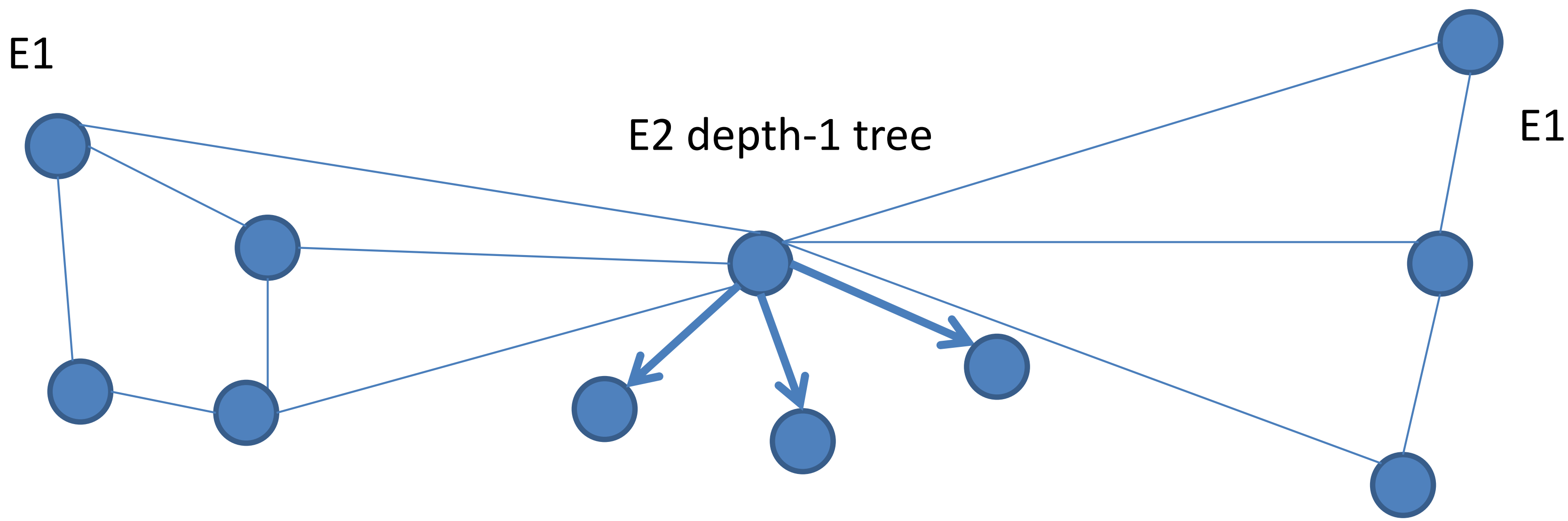


Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.

Claim: does not change connected components.

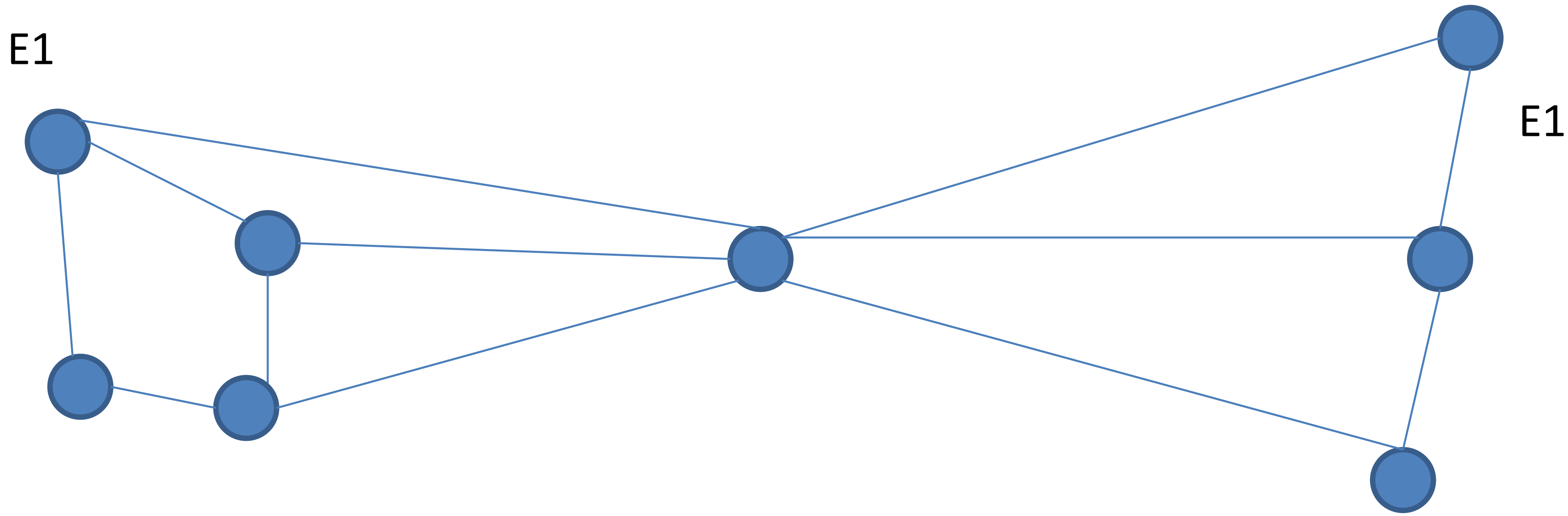


Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.

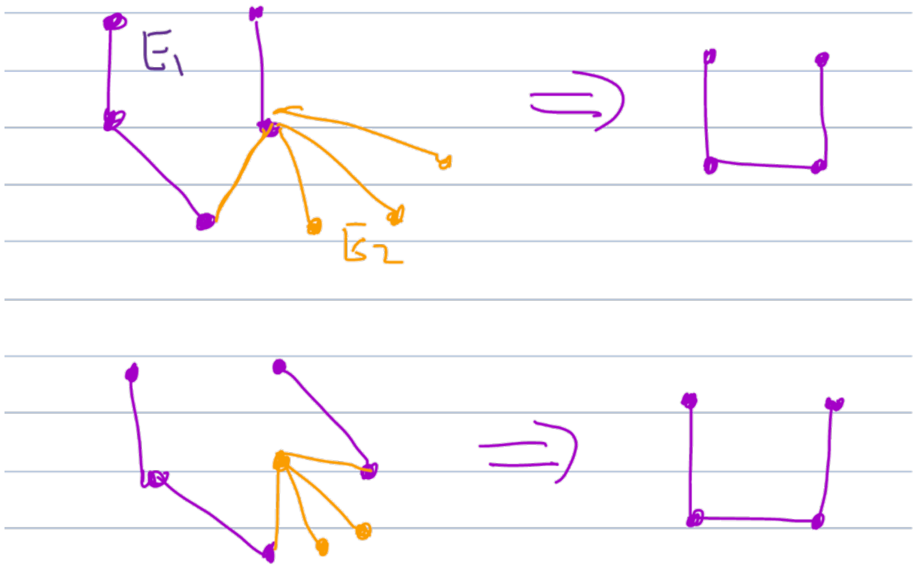
Claim: does not change connected components.



Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.



Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 → depth 1 trees.
3. Contract E1.

Claim: does not change connected components.

Algorithm:

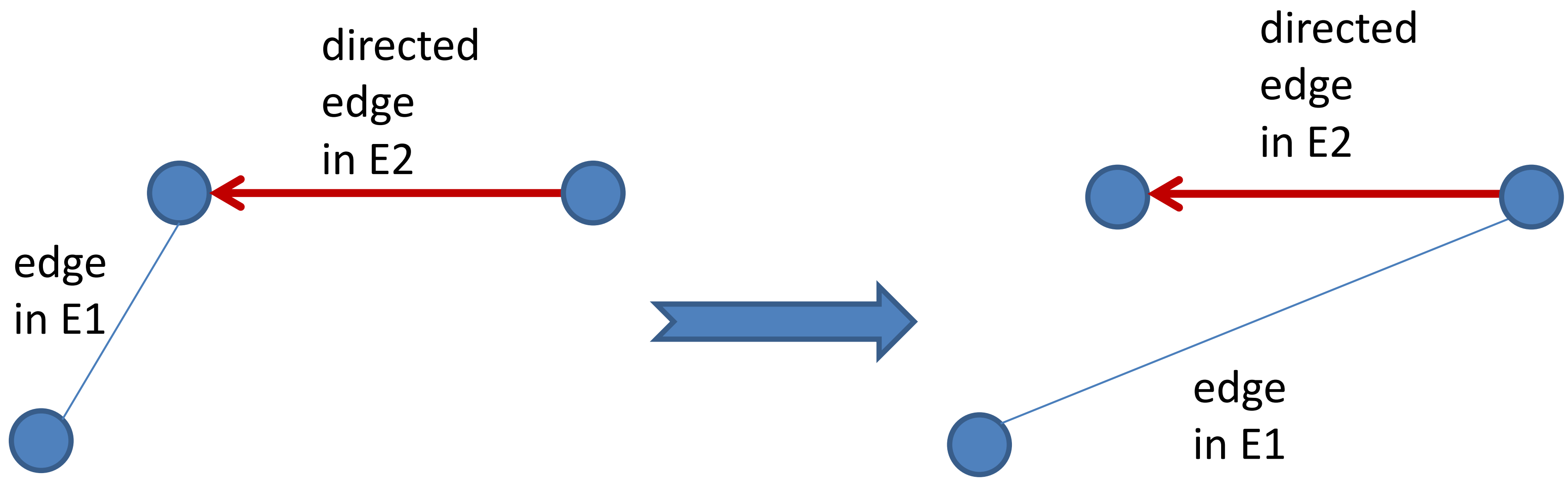
For each (x,y) in E1: if (a,x) or (a,y) is in E2 then:

Replace (x,y) with (y,a) or (x,y) with (x,a).

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.



Only "root" nodes in E2 are connected to E1.

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.

Cache-Efficient Connectivity

Algorithm Idea

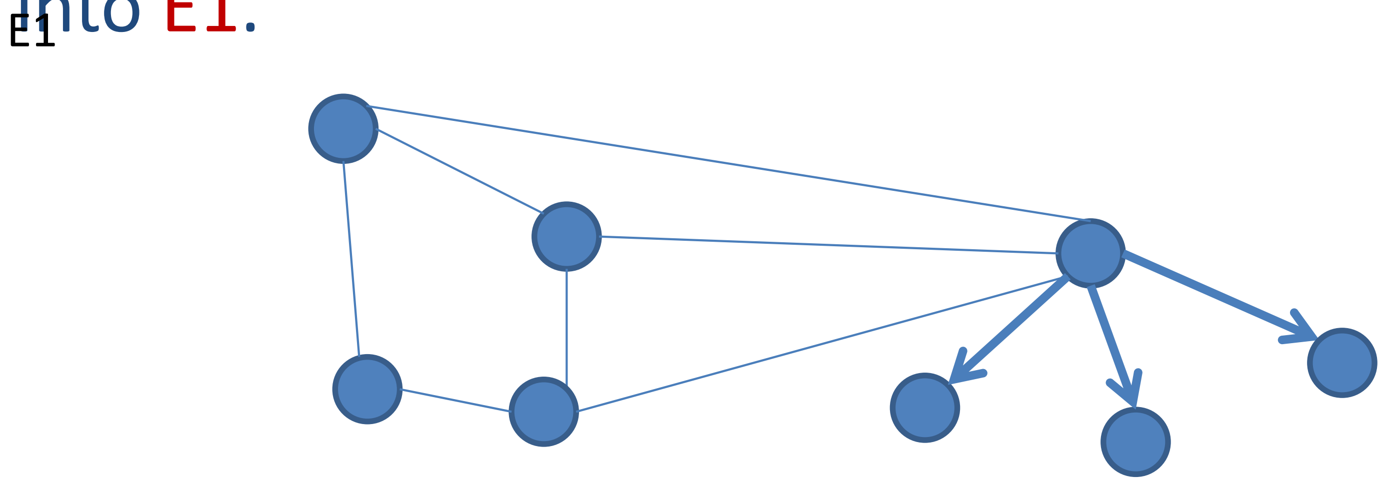
1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 → depth 1 trees.
3. Contract E1.
4. Recursively solve E1 → depth 1 trees.
5. Merge E2 into E1.

E1

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.



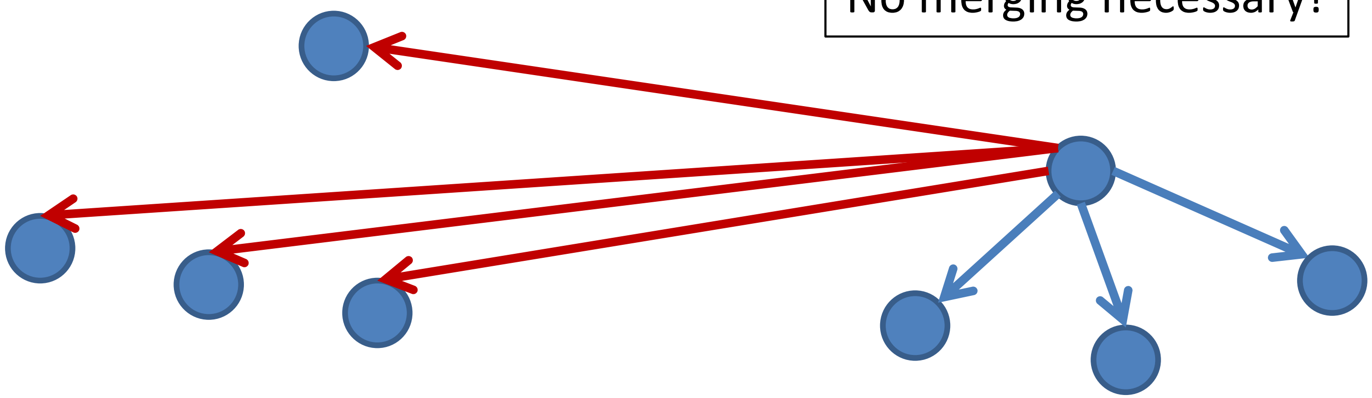
Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.

E1

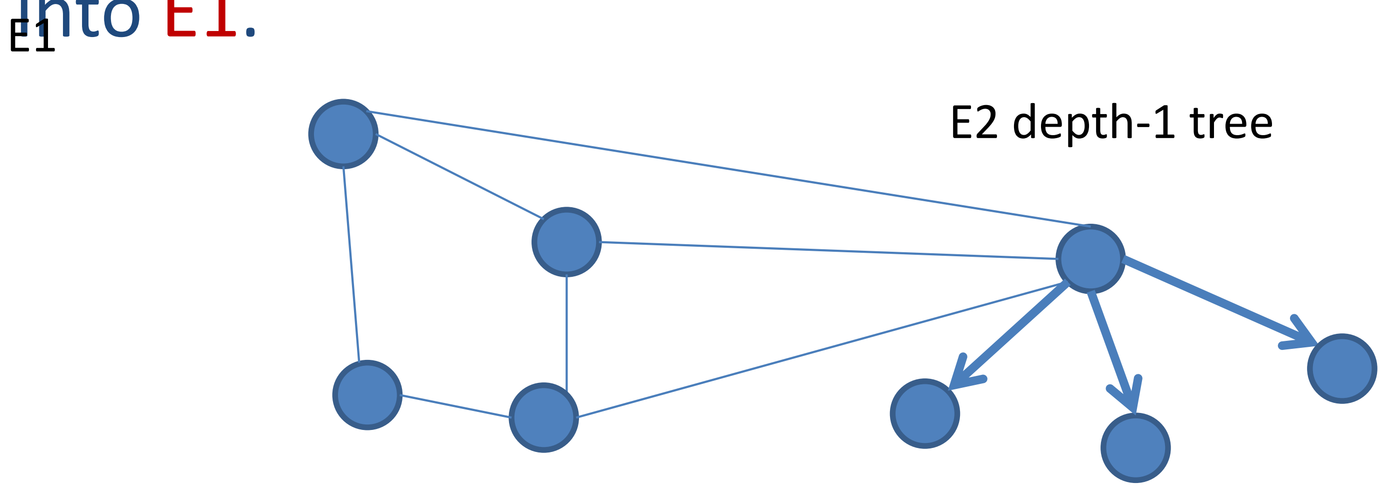
No merging necessary!



Cache-Efficient Connectivity

Algorithm Idea

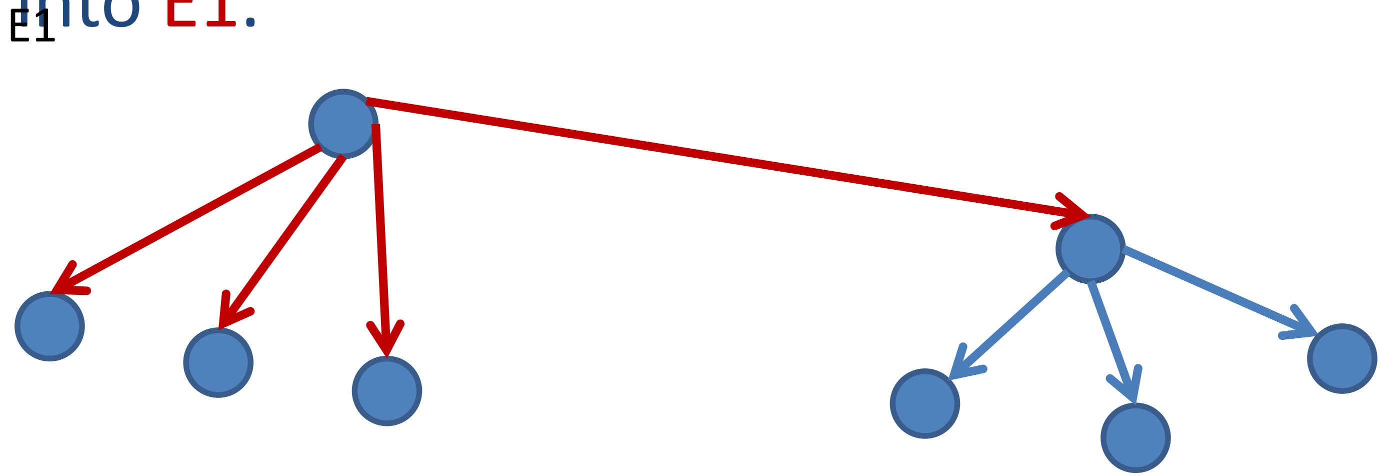
1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.



Cache-Efficient Connectivity

Algorithm Idea

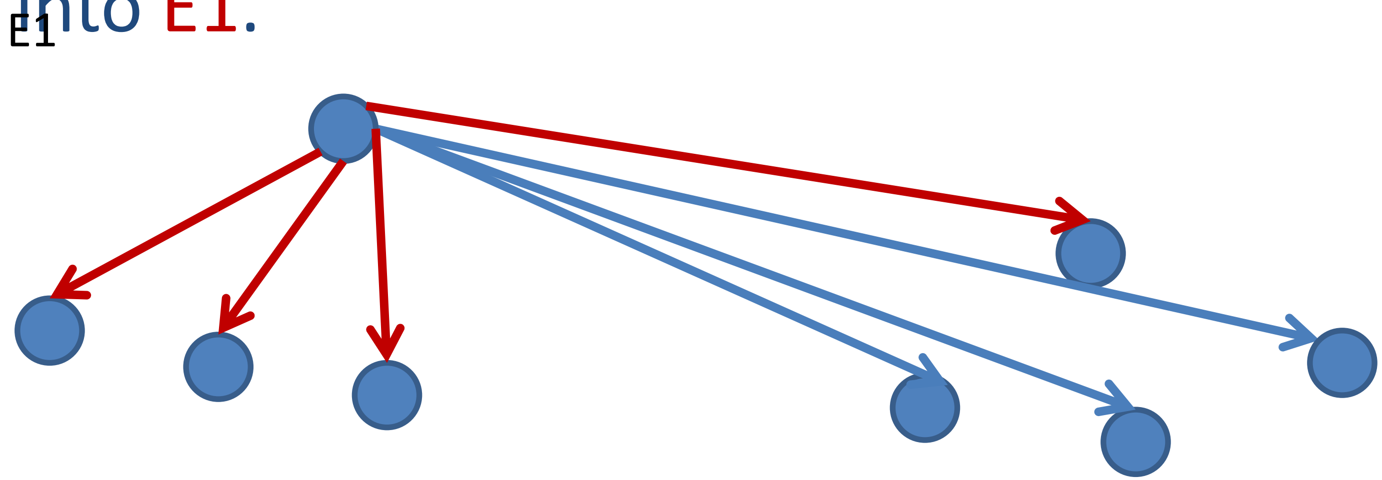
1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.



Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.



Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 → depth 1 trees.
3. Contract E1.
4. Recursively solve E1 → depth 1 trees.
5. Merge E2 into E1.

E1

Claim: Does not change connected components.

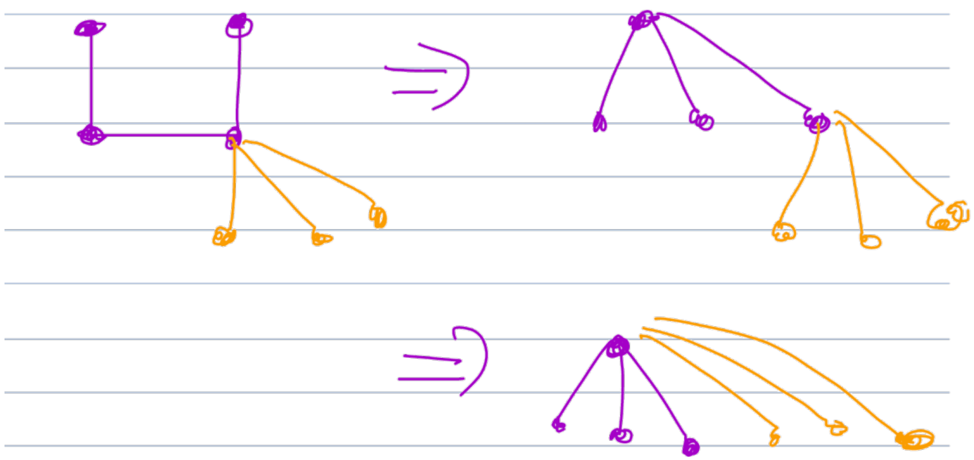
<u>Algorithm:</u>
For each (a,b) in E2:
    If a is an E1 root: add (a,b) to E1.
    Else if (x,a) in E1: add (x,b) to E1.

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.

E1



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 → 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 → delete(a,b), add(x,a)

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)**					

Sort E1 by first, E2 by second.

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)

(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)

(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,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)

(a,b)	(z,d)	(z,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)

(a,b)	**(z,d)**	**(z,c)**	**(c,e)**	**(c,f)**	**(d,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)

(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)**					

Replace…

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)**					

Scan…

Cache Efficient Contract

E1 (sorted by first)

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

E2 (sorted by second)

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

Replace…

Cache Efficient Contract

E1 (sorted by first)

(a,b)	**(z,d)**	**(z,c)**	**(z,e)**	**(z,f)**	**(d,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)

(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)**					

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)

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

E2 (sorted by second)

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

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 $\Rightarrow$ 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)

$$O(sort(E) + E/B)$$

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 → add(a,c) to E1

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

$$O(sort(E) + E/B)$$

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 → depth 1 trees.
3. Contract E1.
4. Recursively solve E1 → depth 1 trees.
5. Merge E2 into E1.

E1

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.

E1

$$O(sort(E) + E/B)$$

Cache-Efficient Connectivity

Algorithm Idea

1. Divide E into two parts: E1 and E2.
2. Recursively solve E2 ➔ depth 1 trees.
3. Contract E1.
4. Recursively solve E1 ➔ depth 1 trees.
5. Merge E2 into E1.

E1

$$
\begin{aligned}
T(E) &= 2T(E/2) + O(E/B) + sort(E) \\
&= O(sort(E)\log(E))
\end{aligned}
$$

Faster than BFS (except in sparse case)!

Summary

Today: Graph Algorithms

Breadth-First-Search

- *Sorting your graph*

MIS

- *Luby's Algorithm*
- *Cache-efficient implementation*

MST

- *Connectivity*
- *Minimum Spanning Tree*

Cache-Efficient MST

Algorithm Idea

1. Let e be a random edge.
2. Divide E into two parts:
   - E1 has edges with weight < w(e).
   - E2.has edges with weight > w(e)
3. Recursively find MST of E1.
4. Do something.
5. Recursively find MST of E2.
6. Do something.