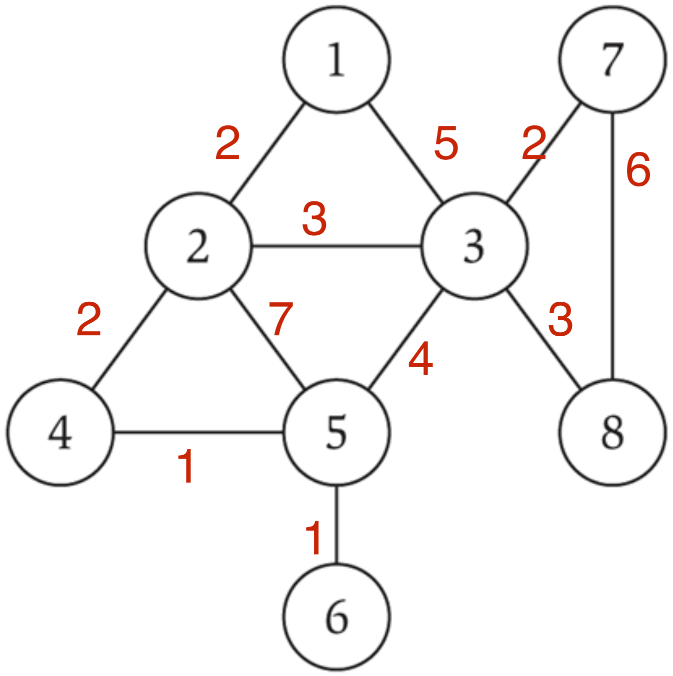
TopHat review



neighbor id

edge weight

NodeId	Neighbors				
1	2 2	3 5			
2	1 2	4 2	5 7	3 3	
3	2 3	1 5	5 4	7 2	8 3
4	2 2	5 1			
8	3 3	7 6			
7	3 2	8 7			
5	2 7	4 1	3 4	6 1	
6	5 1				

Question:

Let A be an adjacency list. What is the total number of keys stored in A?

A. $\Theta(|V|)$ C. $\Theta(|E|)$ E. $\Theta(|V|+|E|)$

B. $\Theta(|V|^2)$ D. $\Theta(\max\{degree(v)\})$

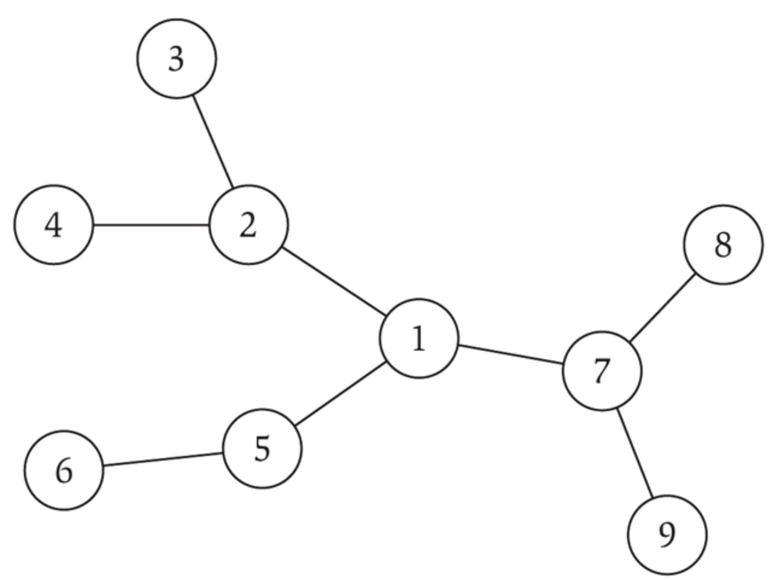
Trees

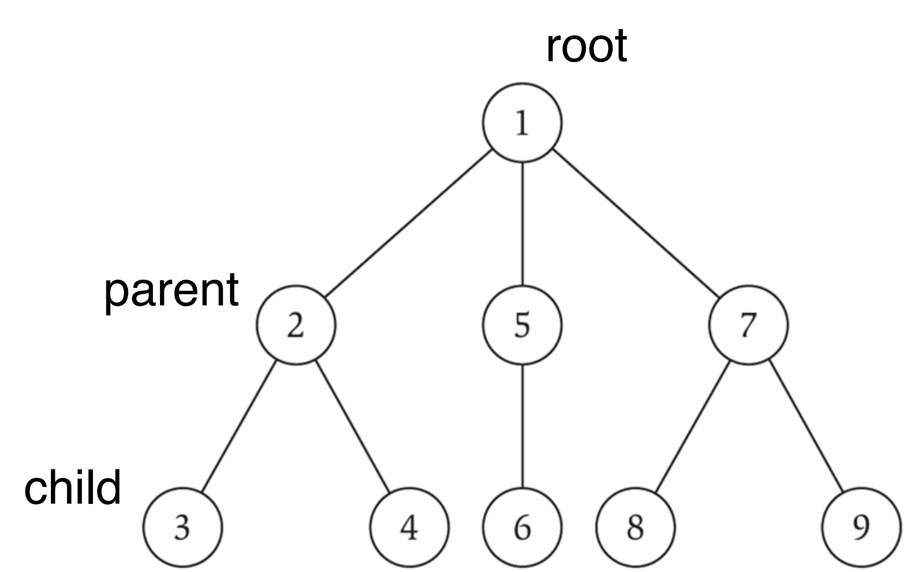
A cycle in a graph is a path, such that the first and last vertex is the same.

An undirected (connected) graph is a tree if it doesn't contain any cycles.

We can designate a node in the tree to be the root.

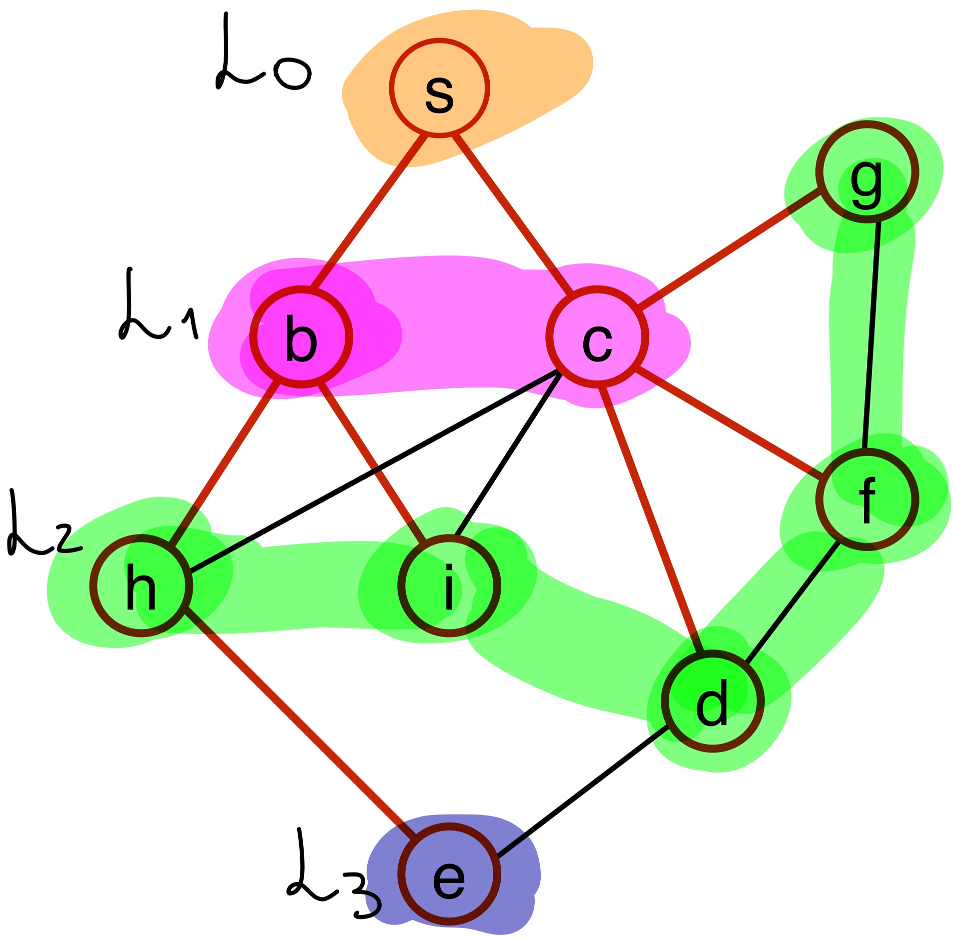
- we use the term parent, child, ancestor in this context. (e.g. parent = the last node on the path from the root to the child.)



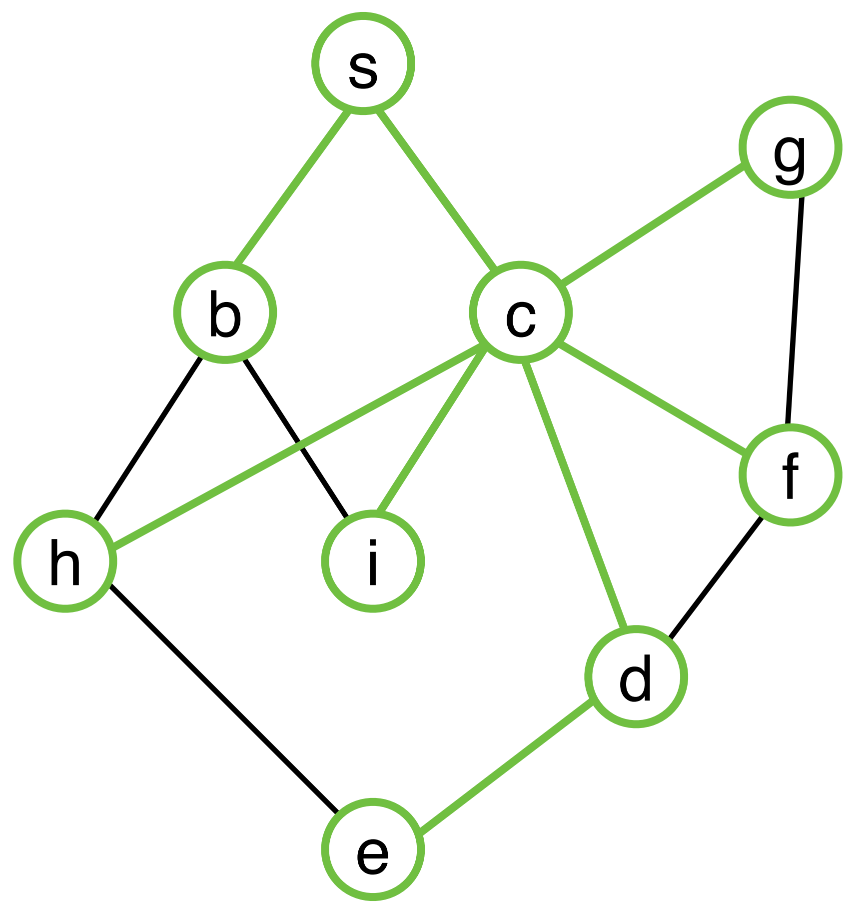


BFS tree

- Trees define a path between nodes.
- What is the relationship between the paths in the BFS tree(s) and the distance of vertices from s?

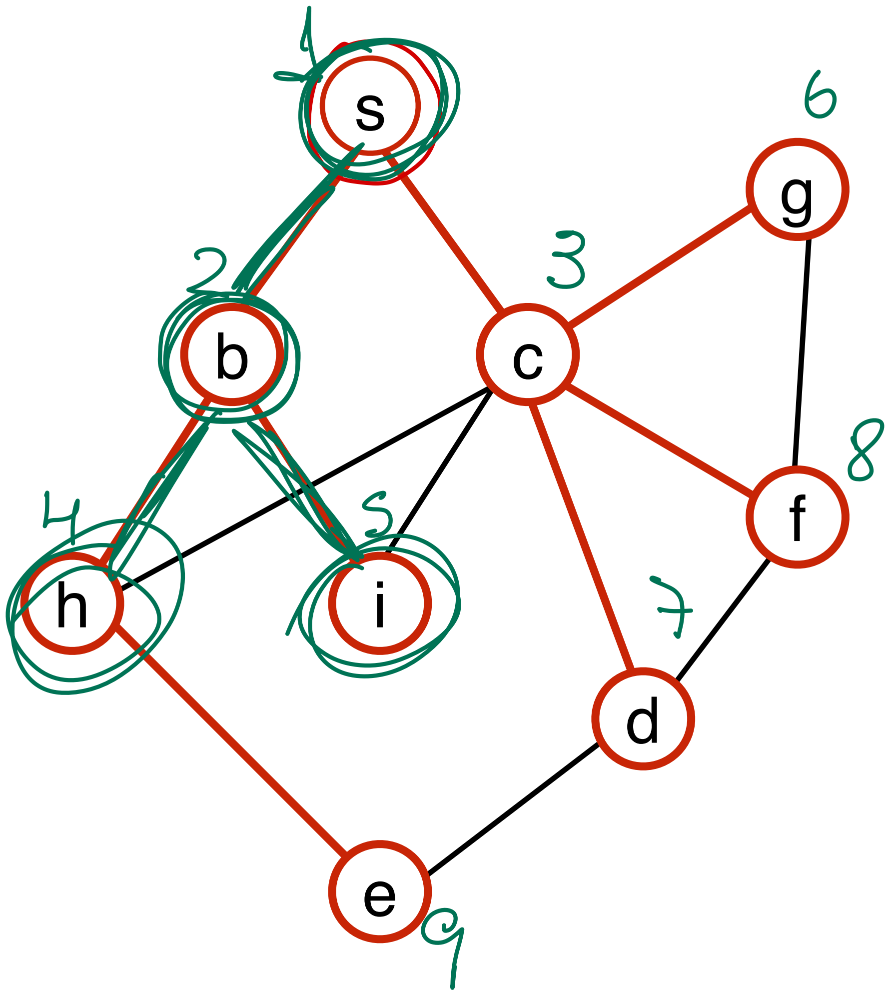


BFS search tree



an other BFS search tree

BFS



Algorithm 1: BFS(G,s)

```
/* G is hash table, the adjacency list of a graph                    */
/* s is a source vertex in G                                          */
parents ← {}/* empty hash table, parents[v] = v's parent.            */
dist ← {}/* empty hash table, dist[v] = distance from s.             */
Q ← empty FIFO queue/* keep track of active nodes                    */
Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization   */  Θ(1)
while Q is not empty do  ← ? Θ(n) times
    u ← Q.dequeue();  Θ(1)
    for v in G[u] do  ← ? δ(u) ←
        /* explore neighbors of active node u                         */
        if v not in parents then
            /* v was so far undiscovered                              */
            parents[v] = u;
            dist[v] = dist[u] + 1;
            Q.enqueue(v);  Θ(1)
return parents, dist
```

notation:

- G[u] returns the neighbor list of node u
- parents[v], dist[v] returns the value (node ID, number) stored at index v

BFS - running time analysis

$\Theta(1)$ = constant time to initialize data structures

Algorithm 1: BFS(G,s)

```
/* G is hash table, the adjacency list of a graph                  */
/* s is a source vertex in G                                        */
parents ← {}/* empty hash table, parents[v] = v's parent.           */
dist ← {}/* empty hash table, dist[v] = distance from s.            */
Q ← empty FIFO queue/* keep track of active nodes                   */
Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization      */
while Q is not empty do
    u ← Q.dequeue();
    for v in G[u] do
        /* explore neighbors of active node u                       */
        if v not in parents then
            /* v was so far undiscovered                            */
            parents[v] = u;
            dist[v] = dist[u] + 1;
            Q.enqueue(v);
return parents, dist
```

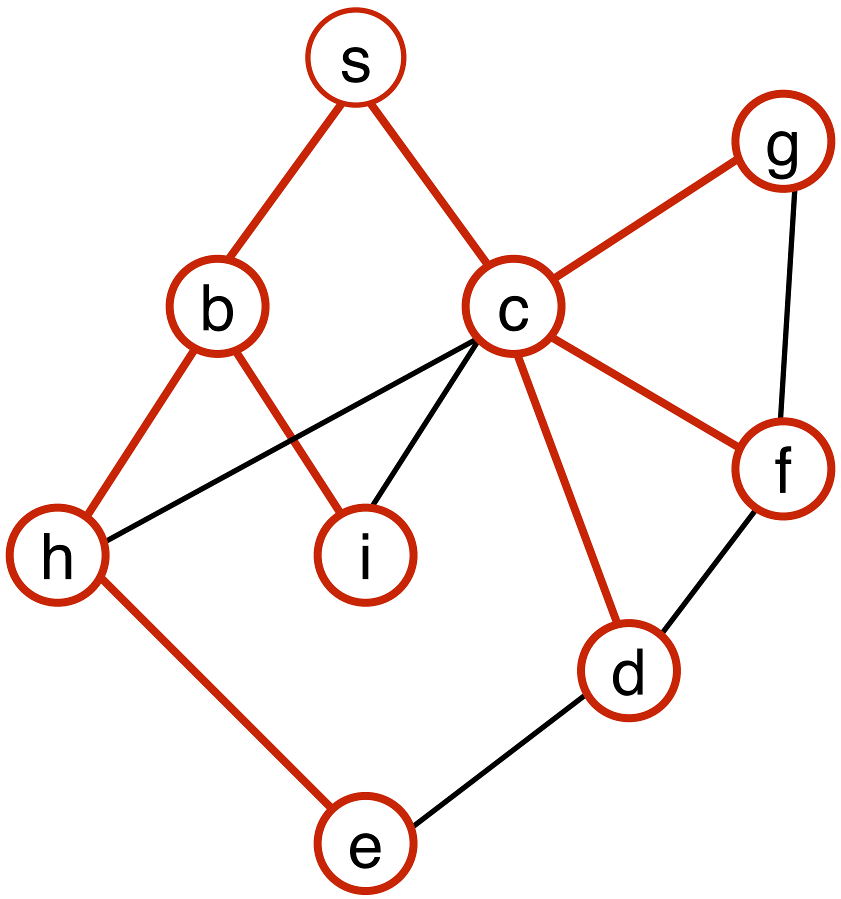
$\Theta(n)$ iterations

$\Theta(\delta(u))$

$$\Theta(n) + \sum_{i=1}^{n} \delta(u) \Rightarrow \Theta(|V| + |E|)$$

BFS

- BFS: a single source shortest paths algorithm, e.g. returns the distance from a node s to each other node v.



BFS search tree

TopHat

Select all that are true when running BFS on graph G from source s.

A. The BFS search tree is unique.
B. When running BFS on G twice with different processing order the values in dist[ ] (i.e. the computed distance values) are identical in the two runs.
C. The shortest path from s to a node v is always unique.
D. If there is an edge connecting nodes u and v then their distances from s cannot be the same.
E. The path from s to v in the BFS tree (i.e. following the edges of the tree from s to v) is a shortest path.

BFS

Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

In consequence BFS should **return**

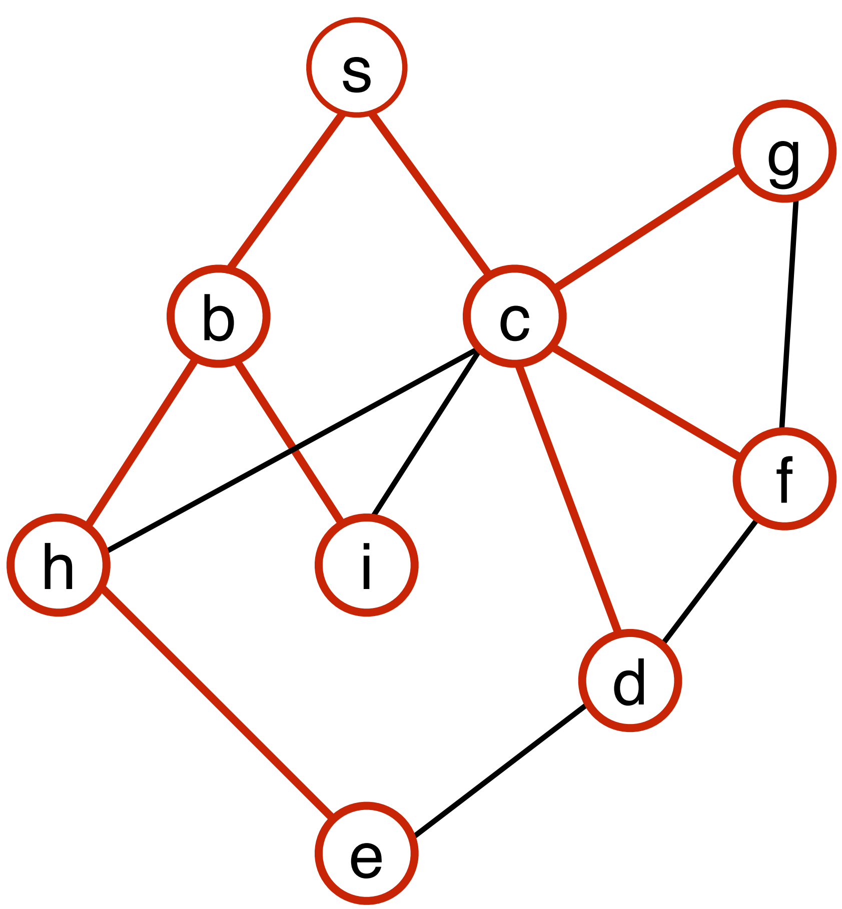
- the layer of each node, so that we get the distances
- the BFS tree, since they encode the shortest paths themselves

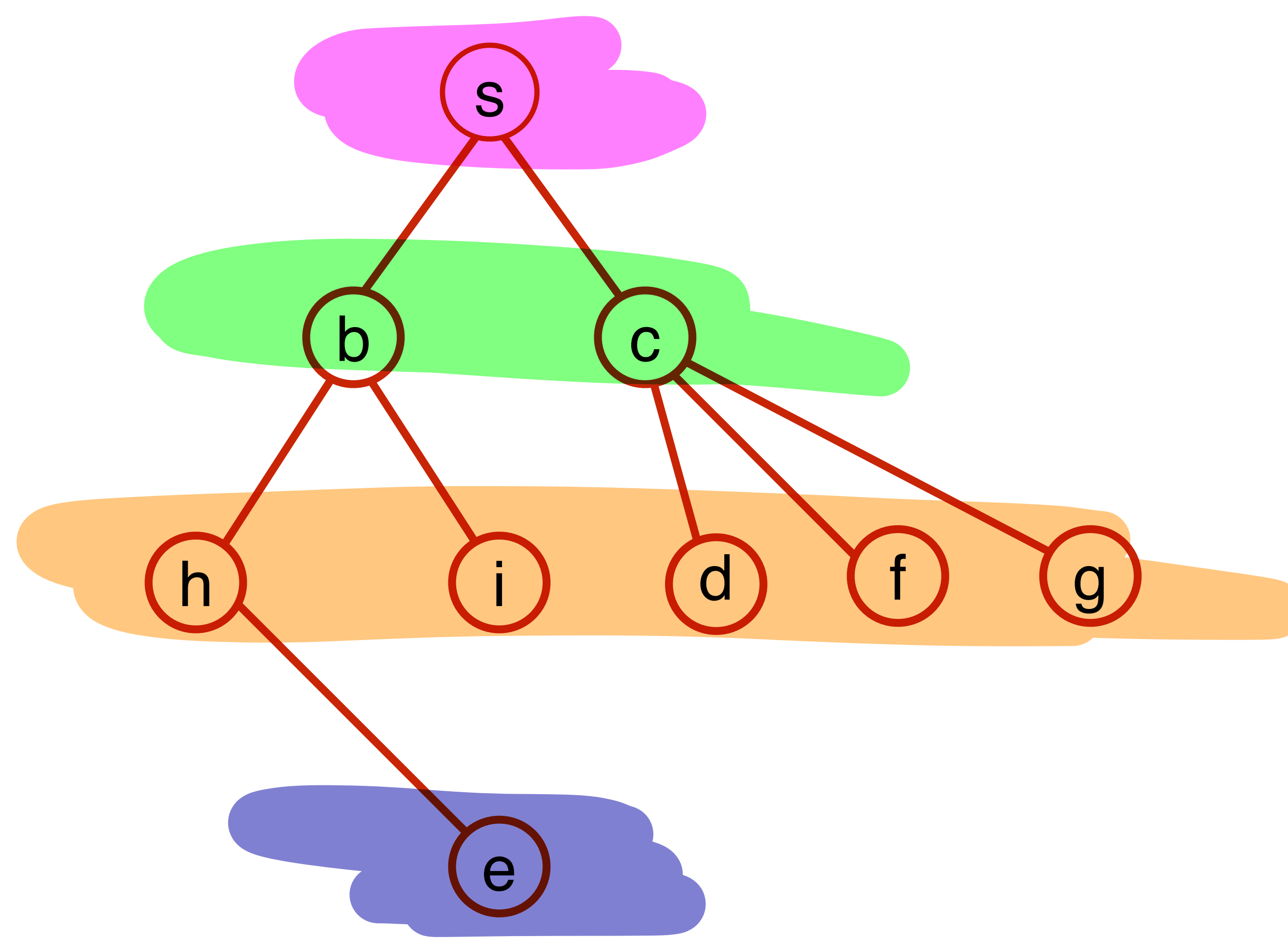
Breadth first search - correctness (multiple slides)

Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

Proof.

This is the BFS tree associated with G.

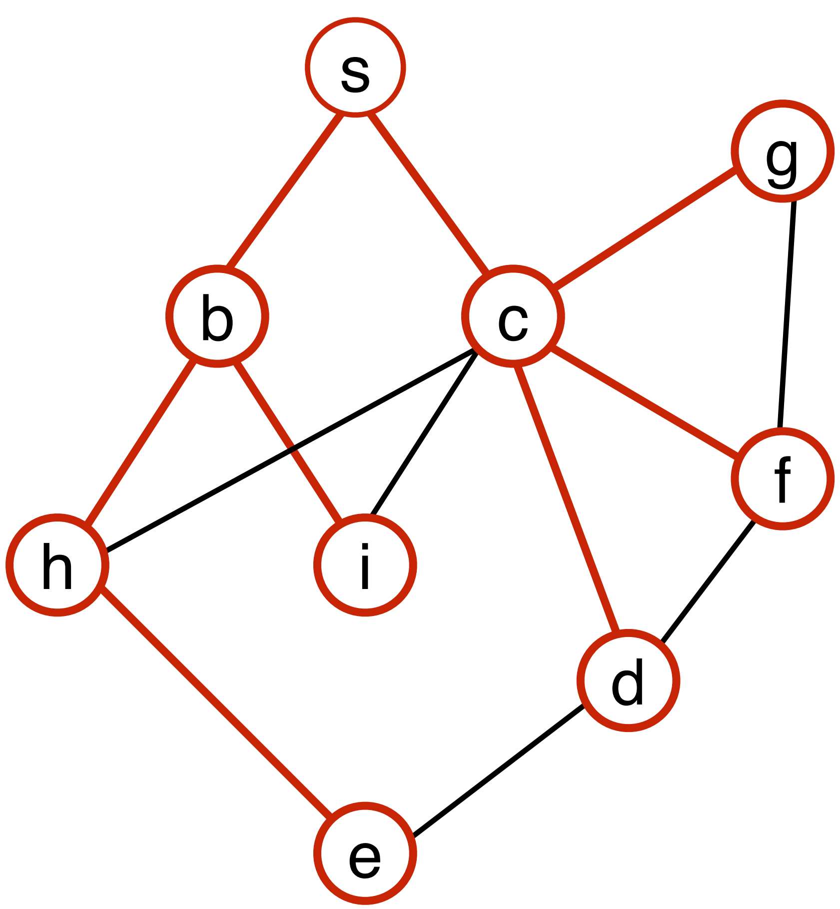


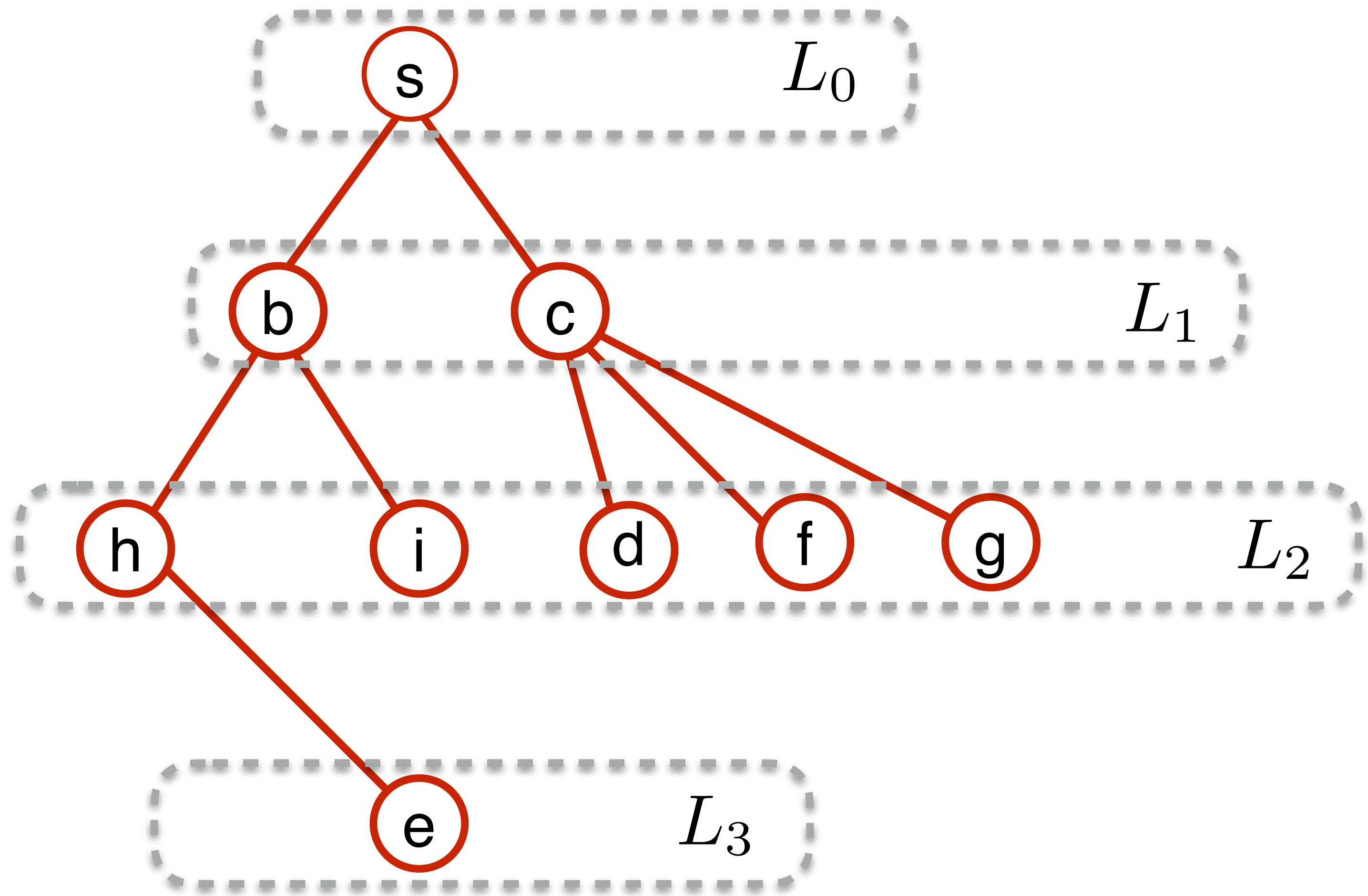


Breadth first search - correctness (multiple slides)

Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

Proof.



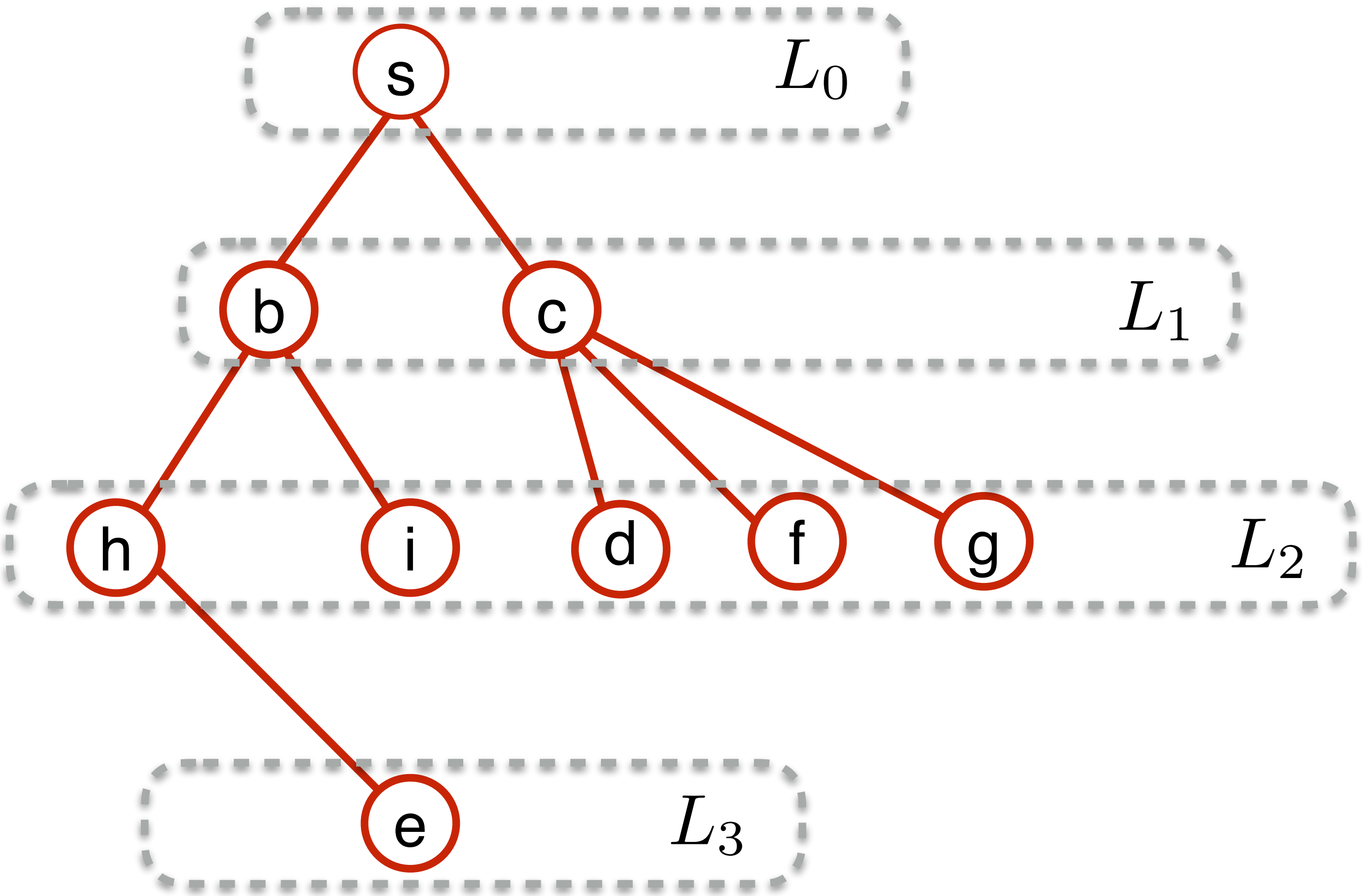


Breadth first search - correctness (multiple slides)

Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

Proof.

Proposition: any two connected nodes are either in the same or consecutive layers.



Note that the proposition applies to every edge in G, both those in the BFS tree and those that are not.

Breadth first search - correctness (multiple slides)

Proposition: any two connected nodes are either in the same or consecutive layers.

Proof:

- For an edge (u,v) without loss of generality we may assume that u was discovered first.
- If v is in the same layer as u, the proposition is true
- if v is not in the same layer, then by the assumption it hasn't been discovered yet
- by the design of BFS v is an undiscovered neighbor of u, hence is assigned to the next layer.

Breadth first search - correctness (multiple slides)

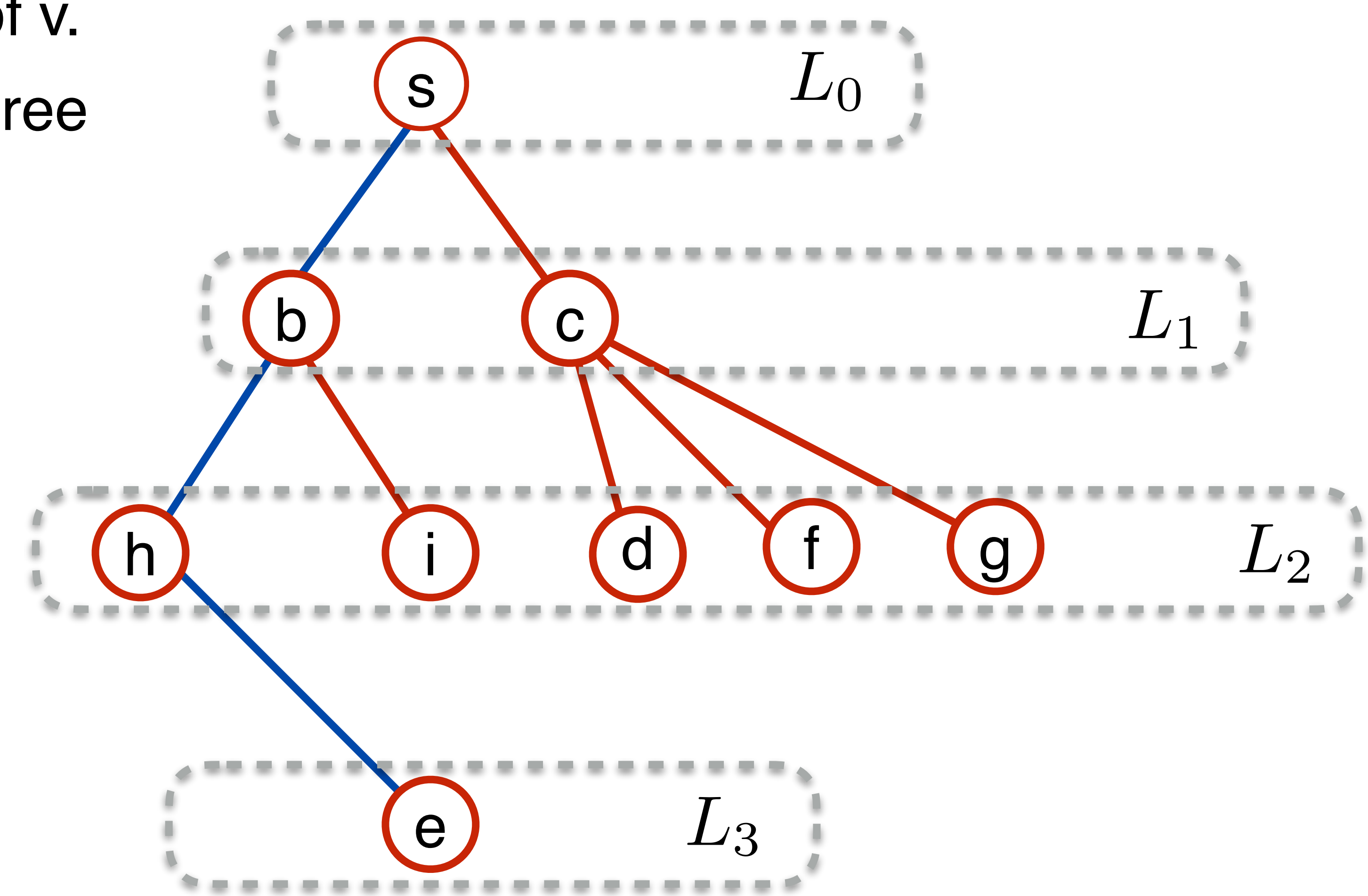
Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

Proof.

Clearly, there is a path from s to v of the same length as the layer of v.

- the one implied by the BFS tree

Is it possible that there is a path from s to v with fewer edges?

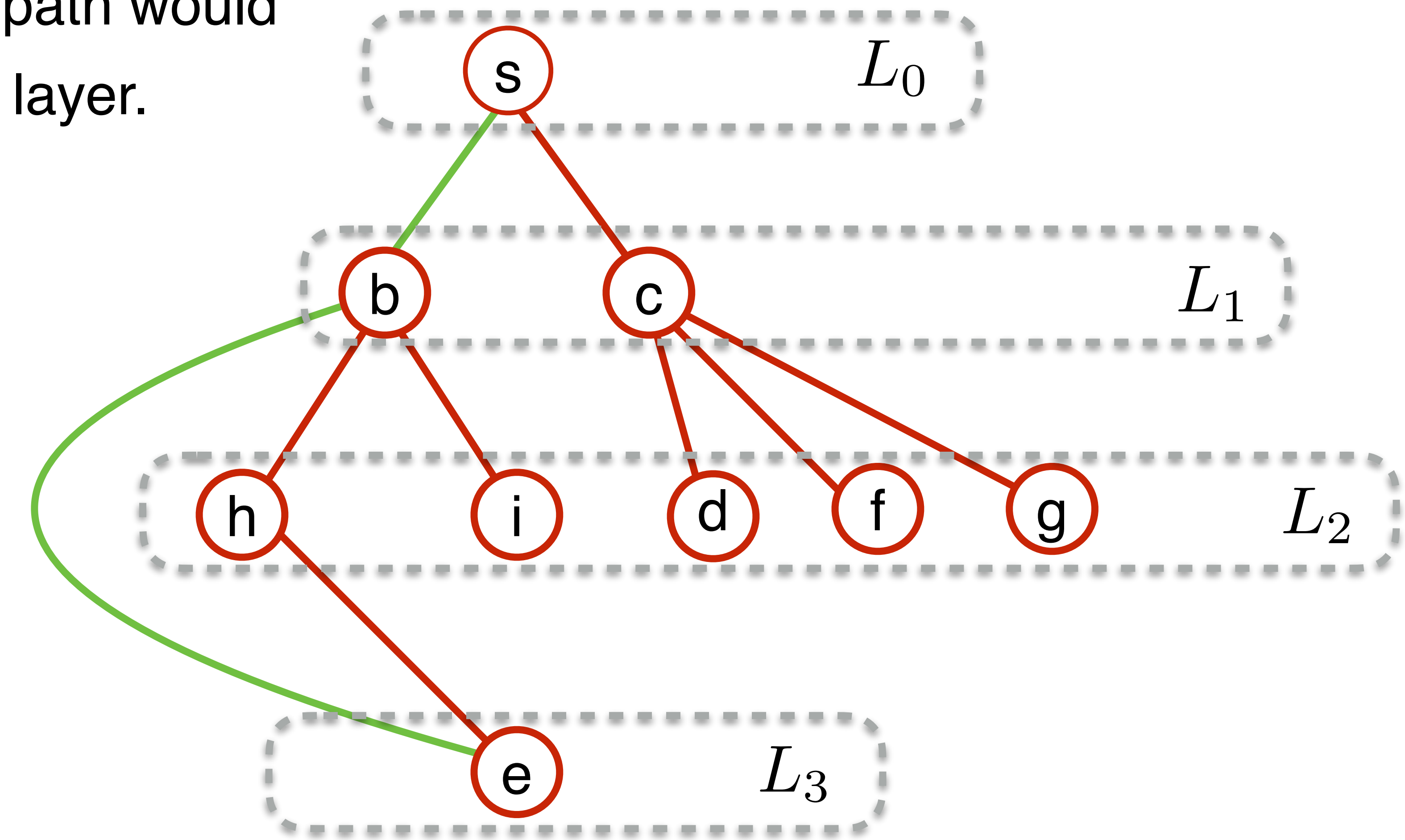


Breadth first search - correctness (multiple slides)

Theorem. For a node v the length of the shortest path connecting s to v is equal to the layer of v in BFS.

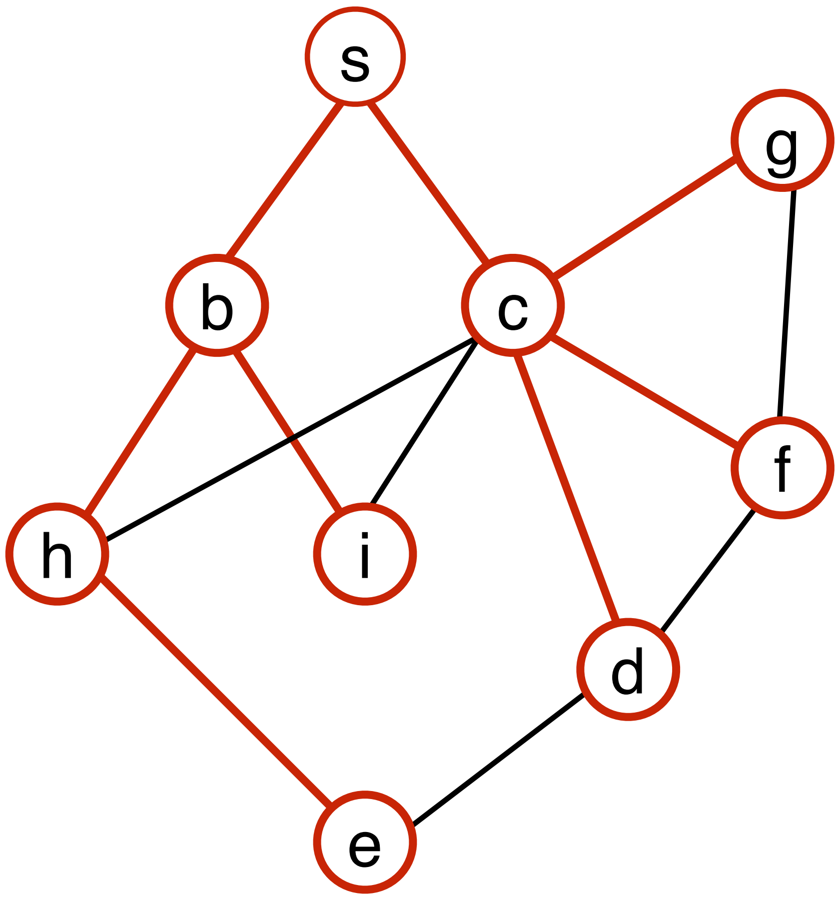
Proof.

For purpose of contradiction, suppose there is a shorter path. That path would have to bypass at least one layer.



This is in contradiction with the proposition that neighbors are at most one layer apart. QED

BFS



Algorithm 1: BFS(G,s)

```
/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */
parents ← {}/* empty hash table, parents[v] = v's parent. */
dist ← {}/* empty hash table, dist[v] = distance from s. */
Q ← empty FIFO queue/* keep track of active nodes */
Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization */
while Q is not empty do
    u ← Q.dequeue();
    for v in G[u] do
        /* explore neighbors of active node u */
        if v not in parents then
            /* v was so far undiscovered */
            parents[v] = u;
            dist[v] = dist[u] + 1;
            Q.enqueue(v);
return parents, dist
```

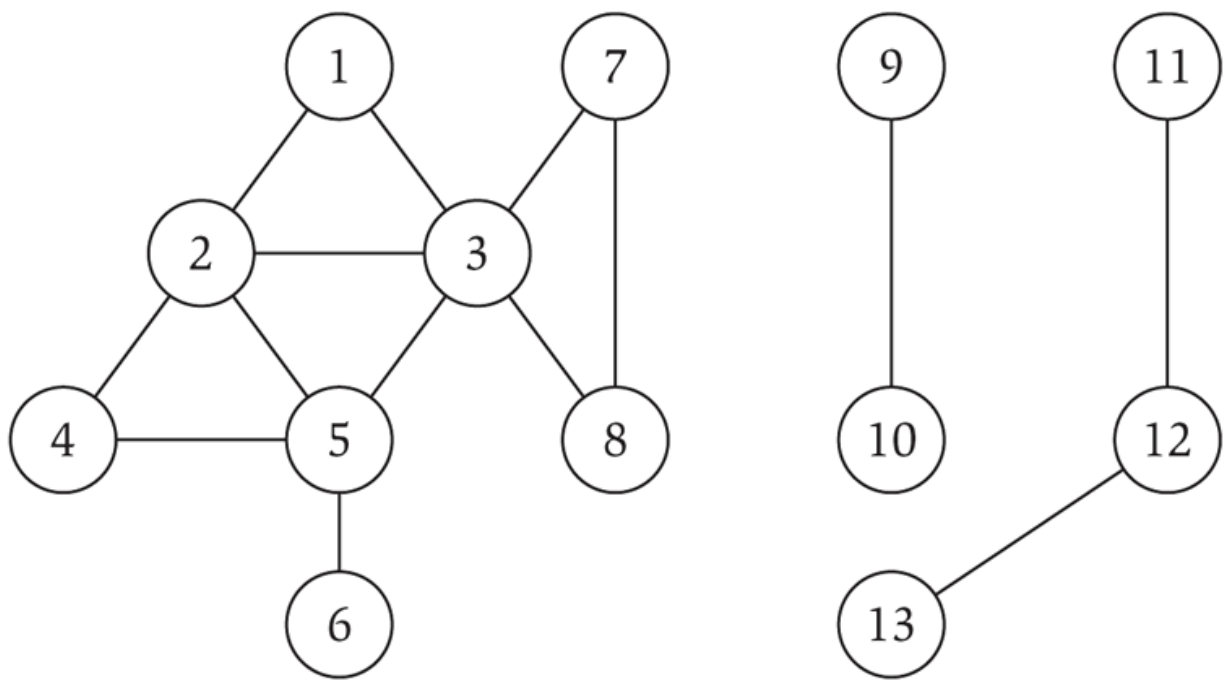
Exercise.

- Given the parents table reconstruct the BFS tree
- For a node v find the path from v back to s. (This is called backtracking)
- do both in O(n) time

Connected component

Def. An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.

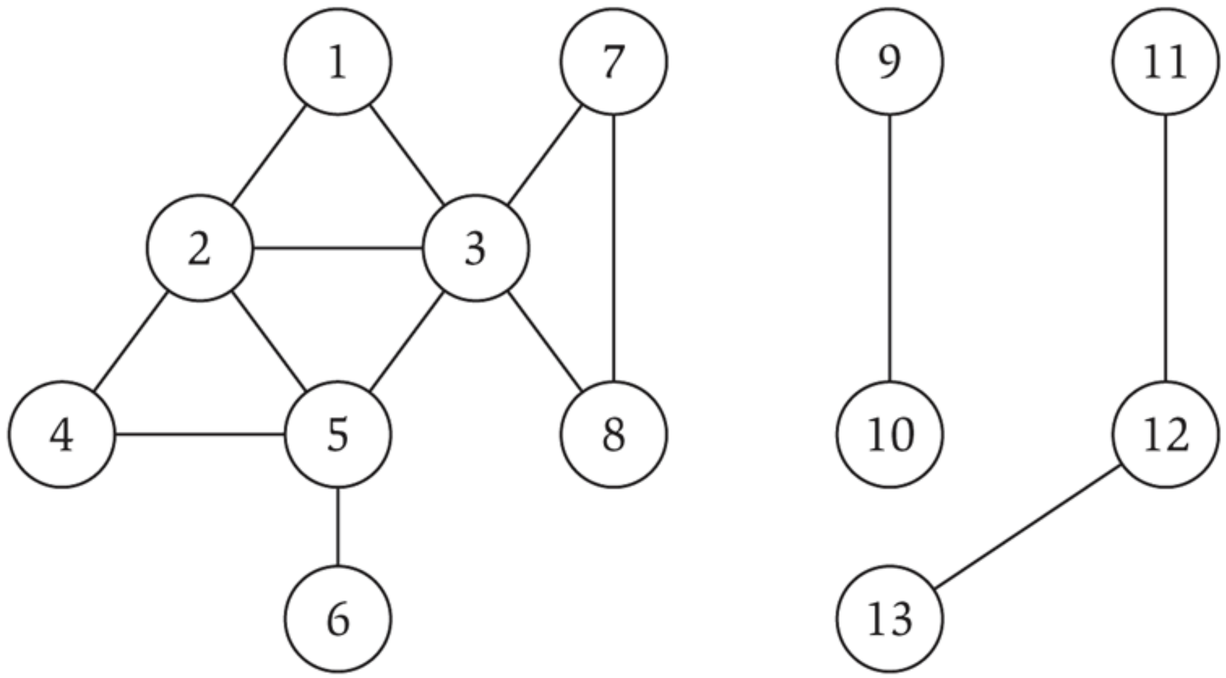
The connected components of a graph are its connected subgraphs.
(an analogous concept for directed graph are the strongly connected components.)



graph with 3 connected components:
{1,2,3,4,5,6,7,8}
{9,10}
{11,12,13}

Connected component

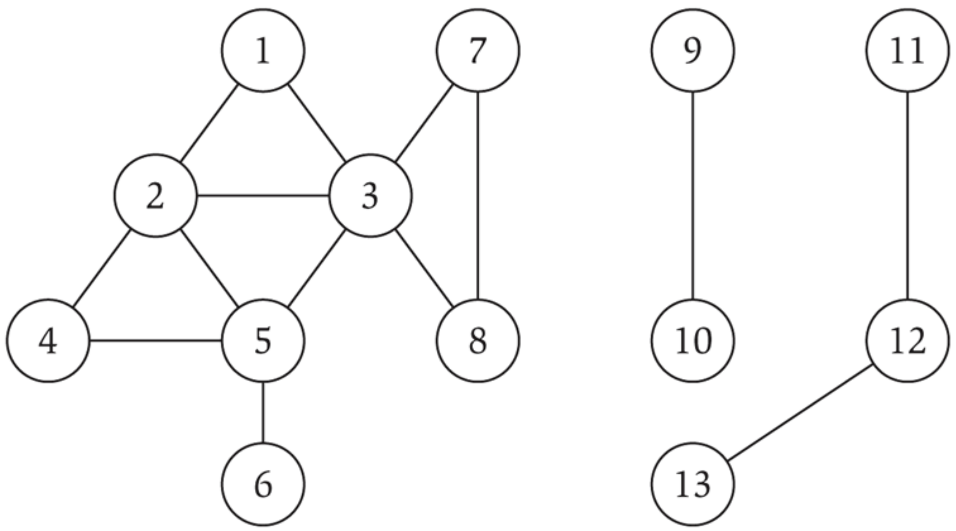
Design an algorithm to find the connected components of an undirected graph.



1. pick a random node v and run BFS from v
   → the nodes that are discovered will be part of v's component
2. pick another w at random that is not discovered yet
3. go back to 1. using w instead of v

Connected component

Find an algorithm to find all the connected components in G.

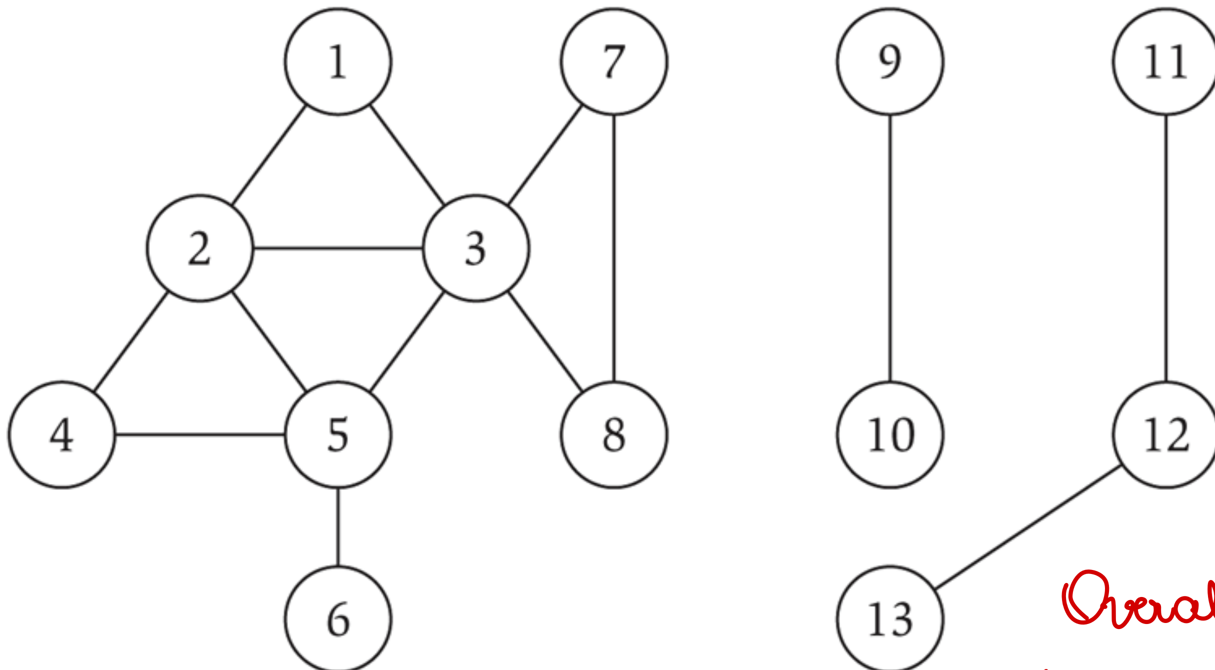


Algorithm: repeatedly run BFS from a yet undiscovered node to find the next connected component.

Running time: still O(n+m)

Connected component - TopHat

Design an algorithm to find the connected components of an undirected graph.



Runtime on component 1

$\Theta(n_1 + m_1)$

Runtime on component 2

$\Theta(n_2 + m_2)$

Runtime on component 3

$\Theta(n_3 + m_3)$

Overall runtime:

$\Theta(n_1+m_1) + \Theta(n_2+m_2) + \Theta(n_3+m_3)$

What is the running time of this algorithm on a graph with n nodes, m edges and k connected components?

$\Theta(\underbrace{n_1+n_2+n_3}+\underbrace{m_1+m_2+m_3})$

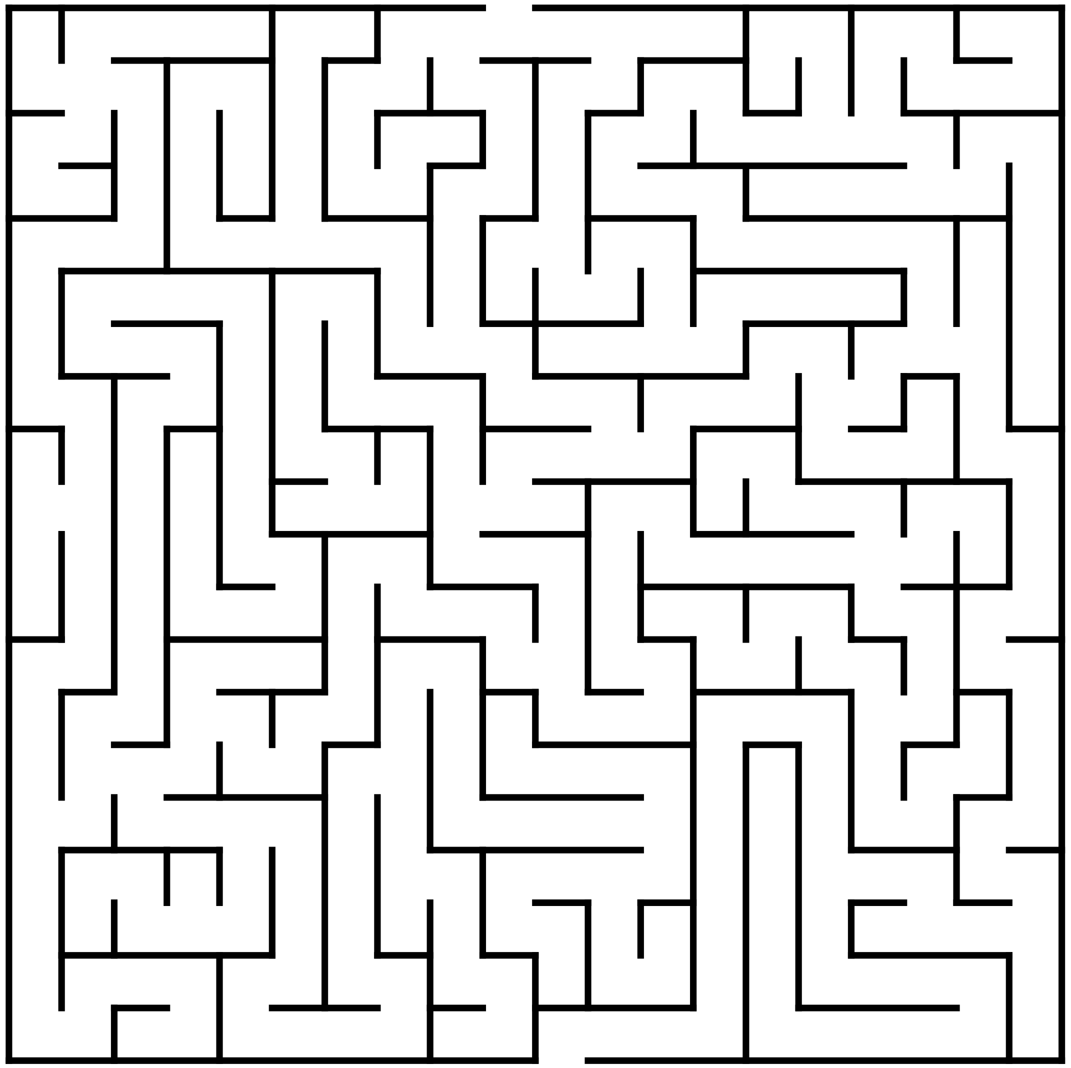
$\Theta(n + m)$

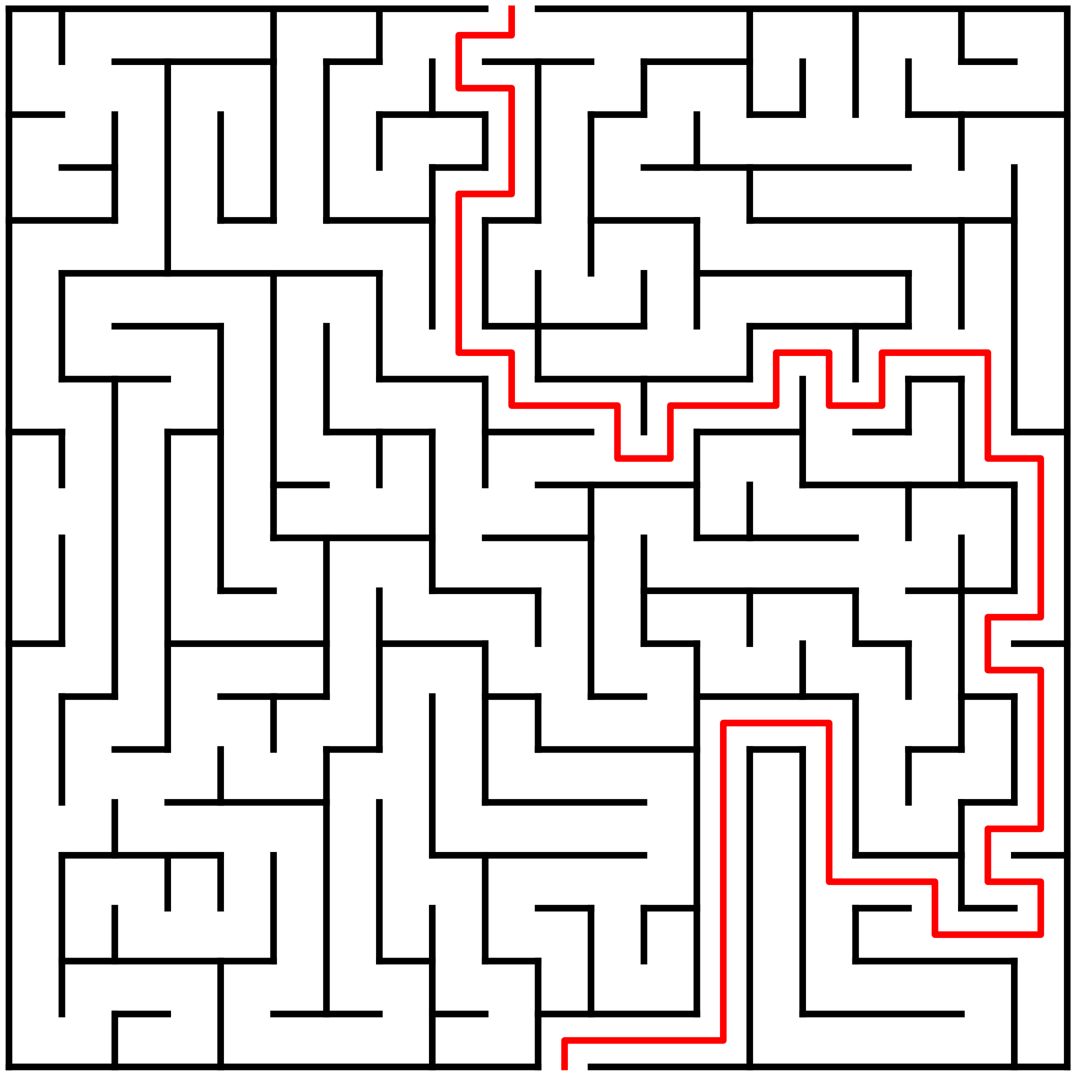
A. $\Theta(n(n+m))$

B. $\Theta(k(n+m))$

C. $\Theta(n+m)$

Maze





Ariadne's thread in logic

Theseus and the Minotaur

- Greek mythology

Ariadne's thread

- principle in logic
- solving a problem through exhaustive application of logic through all available routes
- key element: maintain a record with all available and all exhausted options (the record is sometimes referred to as the 'thread')
- record is kept for the purpose of backtracking - reverse earlier decisions and try alternatives

Depth First Search (DFS) — high level description

Let *G(V,E)* be a directed graph.

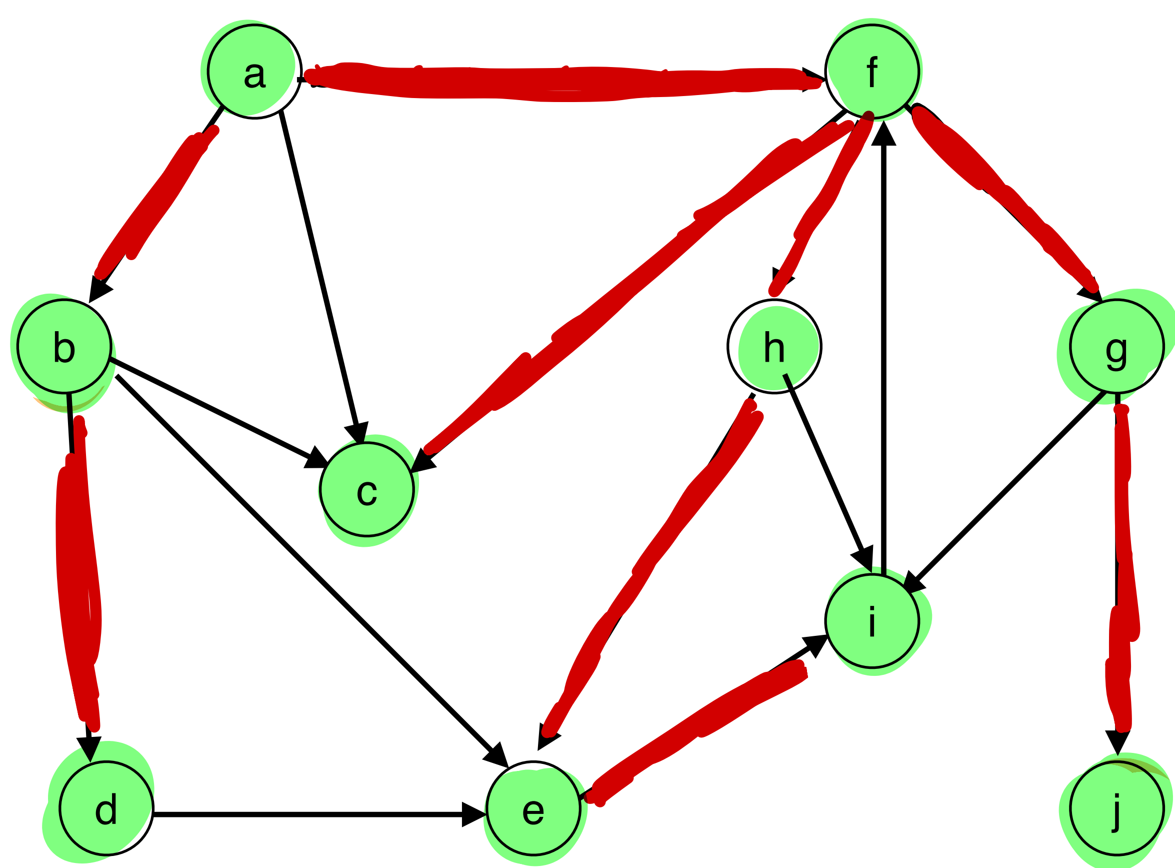
Depth First Search (DFS): Graph traversal algorithm. = it explores the entire graph

- visits all reachable nodes in a graph
- "depth first" — traverse the furthest away from the source first
- recursive in nature

- output:
  - DFS tree
  - timestamps (discovery time, finish time). Used for applications.

(Recommended reading on DFS CLRS chapter 22.3)

DFS



source: a

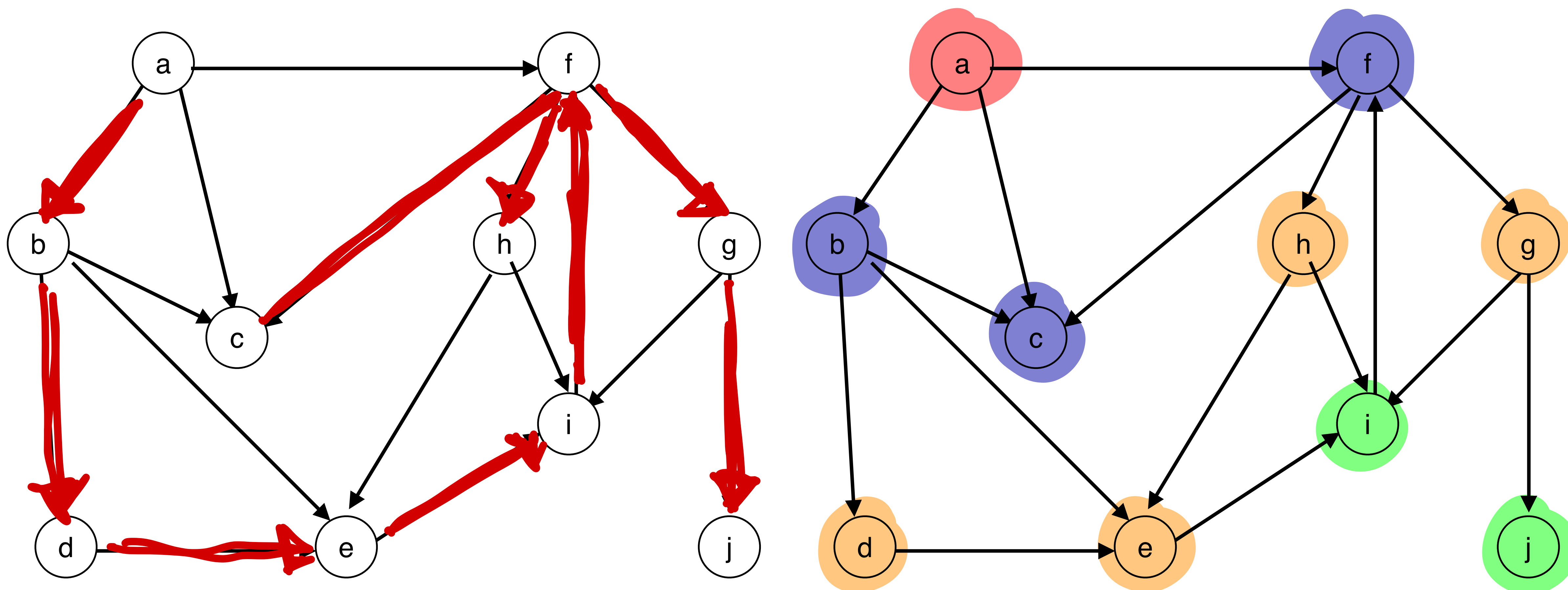
node states: ~~un~~explored, discovered, finished

timestamps:

- discovery time - time of first visit
- finish time - time of last visit

	discovery	finish
a	0	19
b	15	18
c	12	13
d	16	17
e	3	6
f	1	14
g	8	11
h	2	7
i	4	5
j	9	10

DFS vs. BFS example



source: a - b - d - e - i - f

a - f

DFS recursive pseudocode

Algorithm 1: DFSwrapper(G, s)

```
/* G is the adjacency list of a graph */
/* s is a source node */
discovered ← empty set /* ids of discovered nodes */
parents ← hash table /* ids of parents in DFS tree */
times ← hash table/* times[u] = tuple <discovery, finish> */
t ← −1 /* counter */
discovered.add(s), parents[s] = None;
Return DFS(G, s)
```

initialize $\Theta(1)$

Algorithm 2: DFS(G, u)

```
discovered.add(u);
t = t + 1;
times[u][0] = t;
for v in G[u] do
    /* recursively explore u's neighbors */
    if v not in discovered then
        parents[v] = u;
        DFS(G, v);
t = t + 1;
times[u][1] = t;
```

How can we figure out the amount of recursive calls?

Exercise. write and iterative implementation of DFS using stacks.

DFS recursive – runtime of recursive algorithm

Algorithm 1: DFSwrapper(G, s)

/* G is the adjacency list of a graph */
/* s is a source node */
discovered $\leftarrow$ empty set /* ids of discovered nodes */
parents $\leftarrow$ hash table /* ids of parents in DFS tree */
times $\leftarrow$ hash table/* times[u] = tuple <discovery, finish> */
$t \leftarrow -1$ /* counter */
discovered.add(s), parents[s] = None;
Return DFS(G, s)

$\Theta(1)$

Algorithm 2: DFS(G, u)

discovered.add(u);
$t = t + 1$;
times[u][0] $= t$;
for v *in* $G[u]$ **do**
    /* recursively explore u's neighbors */
    if v *not in discovered* **then**
        parents[v] $= u$;
        DFS(G, v);
$t = t + 1$;
times[u][1] $= t$;

Does the depth matter?

DFS recursive – runtime of recursive algorithm

How many recursive calls to DFS total?

One call of DFS for every node in G

$\boxed{\Theta(n) \text{ calls}}$

Runtime of operations done within the current DFS call?

$\Theta(\delta(u))$

Algorithm 2: DFS(G, u)

discovered.add(u);
$t = t + 1$;
times$[u][0] = t$;
for v *in* $G[u]$ **do**
    /* recursively explore u's neighbors */
    if v *not in discovered* **then**
        parents$[v] = u$;
        DFS(G, v);
$t = t + 1$;
times$[u][1] = t$;

$\Theta(1)$ = constant time

for loop runs $\Theta(\delta(u))$ times

$\Theta(1)$

→ n recursive calls

$$\Theta(n) + \sum_{i=1}^{n} \Theta(\delta(u)) = \Theta(|V| + |E|)$$