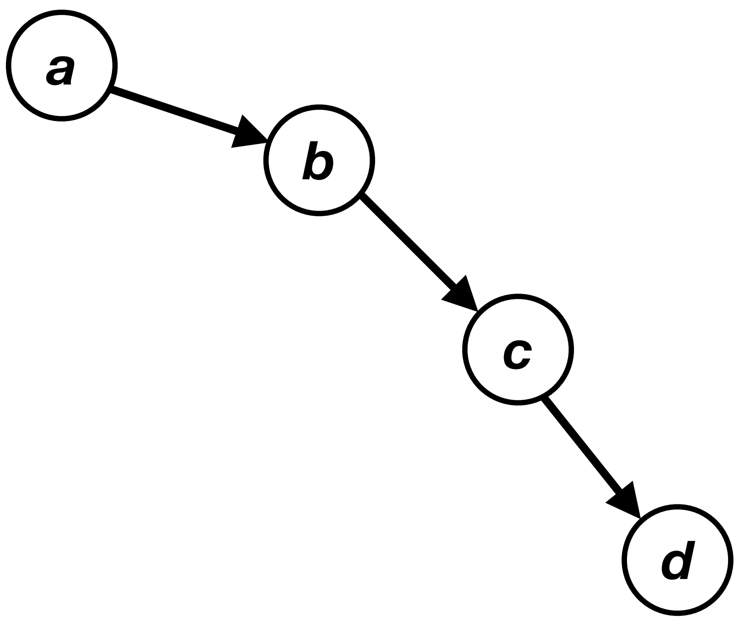
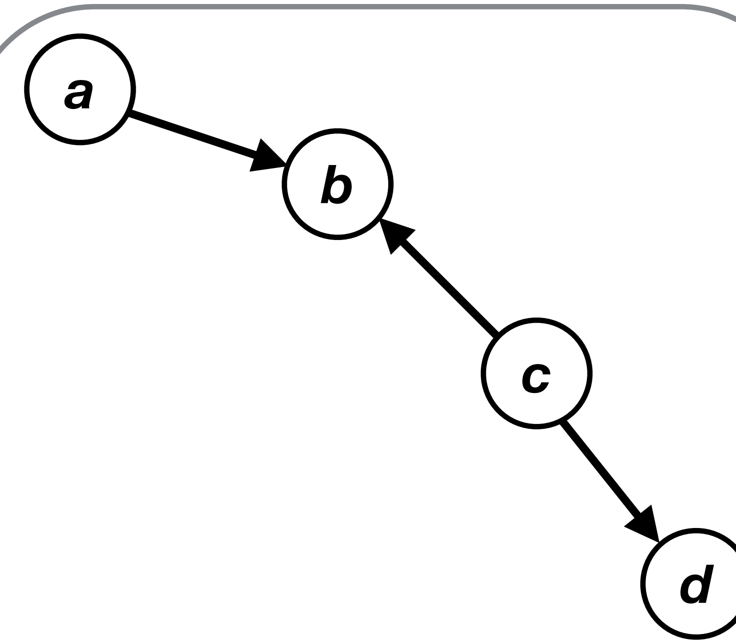
COMP3251
Lecture 8: Breadth-First Search
(Chapter 4.1 and 4.2)

Some Definitions

A **path** is a sequence of edges in which the end vertex of an edge equals the start vertex of the following edge.



A path from *a* to *d*

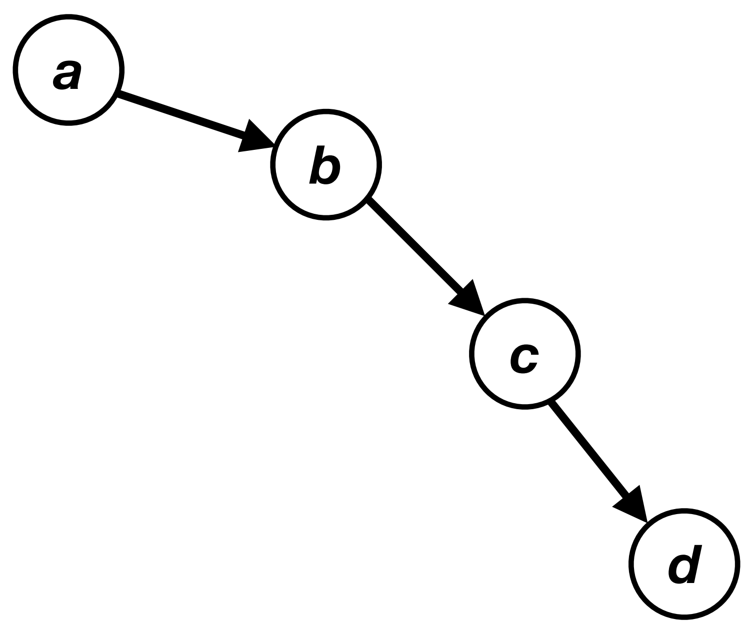


Not a path from *a* to *d*, the directions of the edges are not correct

Some Definitions

A **path** is a sequence of edges in which the end vertex of an edge equals the start vertex of the following edge.

The **length** of a path is the number of edges in this path.



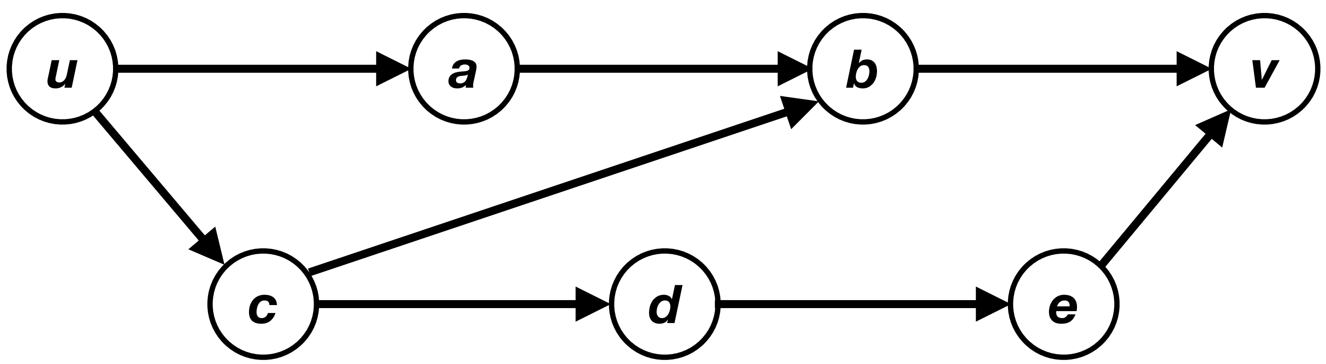
A path from *a* to *d* that has length *3*

Some Definitions

A **path** is a sequence of edges in which the end vertex of an edge equals the start vertex of the following edge.

The **length** of a path is the number of edges in this path.

The **distance** $d(u, v)$ from u to v is the length of the shortest path (the one with the smallest length) from u to v.



There are three paths from u to v, and the shortest ones have length three. Thus, $d(u,v) = 3$.

Single-Source Shortest Paths Problem

Given an input directed graph $G = (V, E)$ and a specific vertex $s \in V$, find, for every vertex $x \in V$, the distance from s to x.

Single-Source Shortest Paths Problem

Given an input directed graph $G = (V, E)$ and a specific vertex $s \in V$, find, for every vertex $x \in V$, the distance from s to x.

Assumption (for simplicity): All vertices are reachable from s.

Notations:

- For any vertex v, let $dist(v) = d(s, v)$.
- For any $k \geq 0$, let L_k be the set of vertices v with $dist(v) = k$.
- Let $adj(L_k)$ be the set of vertices that are adjacent to some vertices in L_k, i.e.,

$$adj(L_k) = \{ u : (v, u) \in E \text{ for some } v \in L_k \}$$

- In essence, our problem is to find L_k for all $k \geq 0$.

A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices *v*.
  As soon as we know that *v* is in L_i, we set *visited(v)* = *true*.

A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices *v*.
  As soon as we know that *v* is in L_i, we set *visited(v)* = *true*.
- We already know L_0, which is { *s* }, and *visited(s)* = *true*.

A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices v.
  As soon as we know that v is in L_i, we set *visited(v)* = *true*.
- We already know L_0, which is $\{ s \}$, and *visited(s)* = *true*.
- We know that L_1 is a subset of $adj(L_0)$, and in general for any $k \geq 0$, L_{k+1} is a subset of $adj(L_k)$.

A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices *v*.
  As soon as we know that *v* is in L_i, we set *visited(v)* = *true*.
- We already know L_0, which is { *s* }, and *visited(s)* = *true*.
- We know that L_1 is a subset of $adj(L_0)$, and in general for any $k \geq 0$, L_{k+1} is a subset of $adj(L_k)$.
- **Key question:** Given L_k, how can we find L_{k+1} from $adj(L_k)$?

A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices *v*.
  As soon as we know that *v* is in L_i, we set *visited(v)* = *true*.
- We already know L_0, which is { *s* }, and *visited(s)* = *true*.
- We know that L_1 is a subset of *adj*(L_0), and in general for any $k \geq 0$, L_{k+1} is a subset of *adj*(L_k).
- **Key question:** Given L_k, how can we find L_{k+1} from *adj*(L_k)?
- **Observation:** Suppose we have determined L_0, L_1, … , L_k, but not L_{k+1}, L_{k+2}, … Then, for any *v* in *adj*(L_k),
  - if *visited(v)* = *true*, then *v* is in L_i for some i ≤ k;
  - if *visited(v)* = *false*, then *v* is in L_{k+1}.

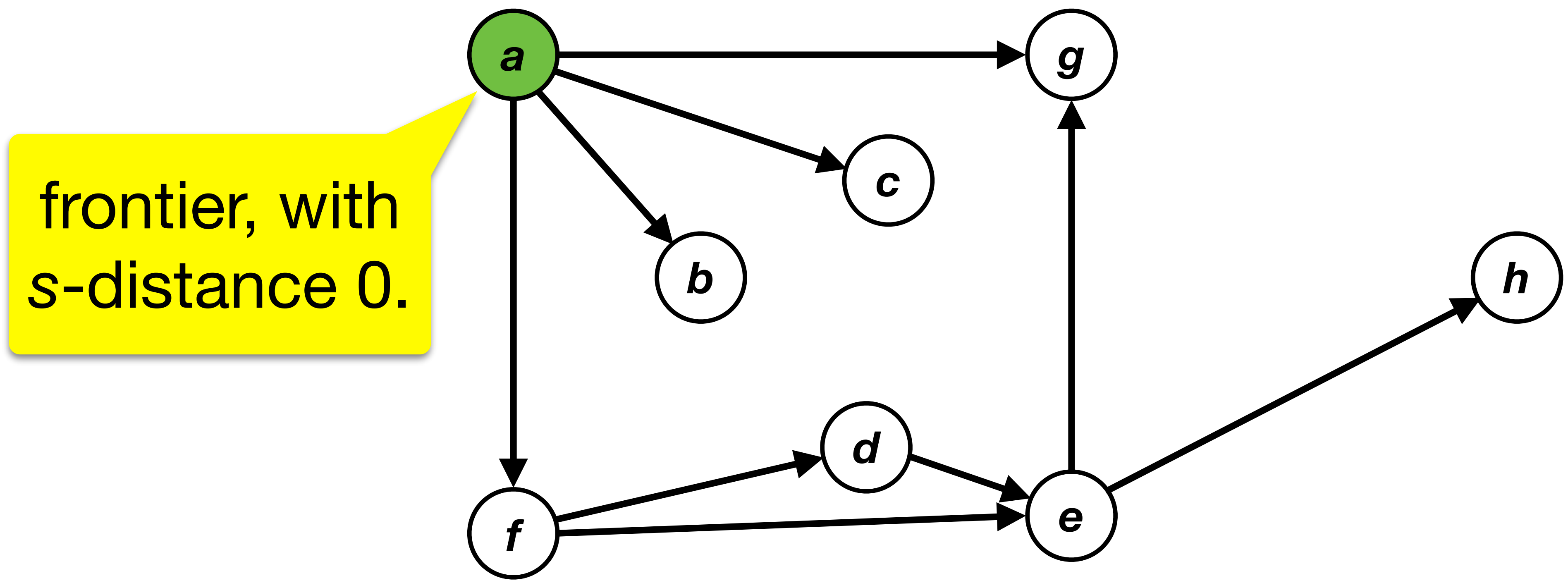
A Simple Idea for Solving the Problem

- Initially, we set *visited(v)* = *false* for all vertices *v*.
  As soon as we know that *v* is in L_i, we set *visited(v)* = *true*.
- We already know L_0, which is { *s* }, and *visited(s)* = *true*.
- We know that L_1 is a subset of *adj*(L_0), and in general for any $k \geq 0$, L_{k+1} is a subset of *adj*(L_k).
- **Key question:** Given L_k, how can we find L_{k+1} from *adj*(L_k)?
- **Observation:** Suppose we have determined L_0, L_1, … , L_k, but not L_{k+1}, L_{k+2}, … Then, for any *v* in *adj*(L_k),
  - if *visited(v)* = *true*, then *v* is in L_i for some i ≤ k;
  - if *visited(v)* = *false*, then *v* is in L_{k+1}.
- Hence, given L_0, we can find L_1 by picking vertices *v* in *adj*(L_0) with *visited(v)* = *false*; and then similarly find L_2, L_3 …

Breadth-First Search (BFS)

Breadth-First Search implements the idea directly. Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier (just like water-front), i.e., the algorithm discovers all vertices in L_k before discovering any vertices L_{k+1}.

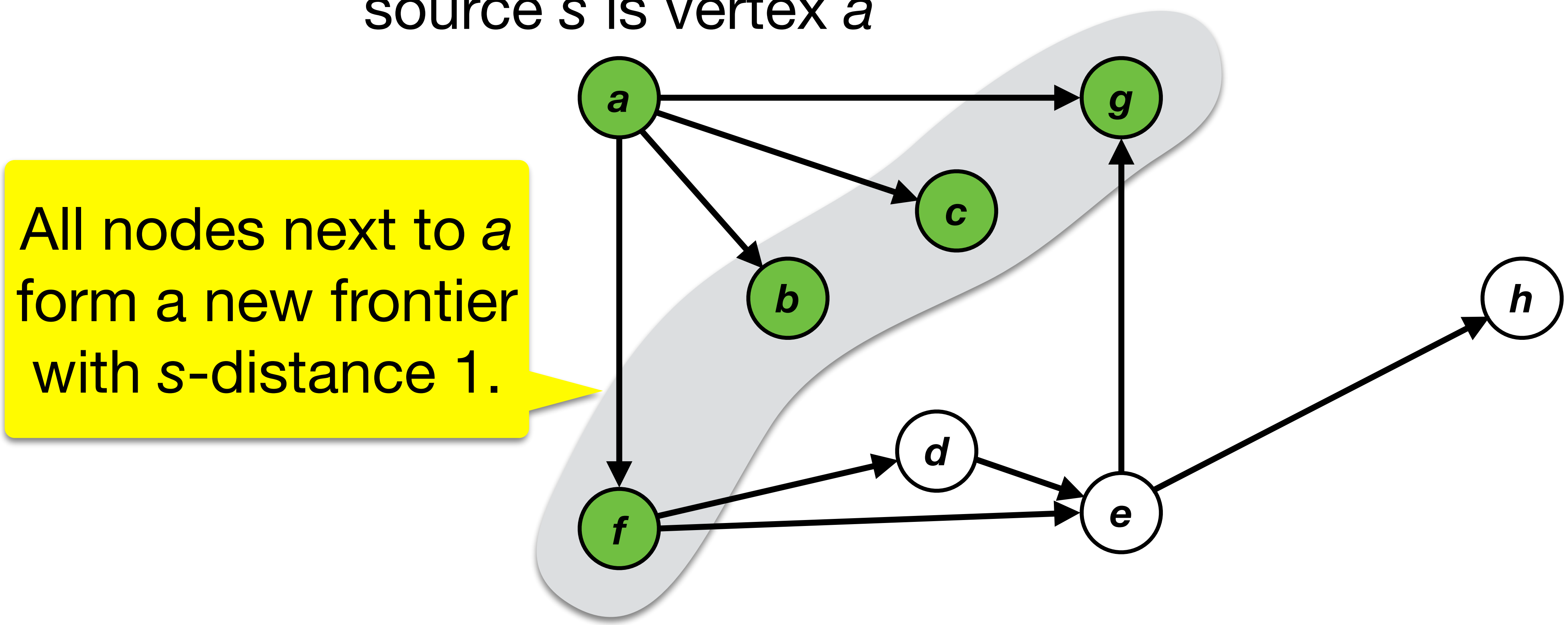
source *s* is vertex *a*



Breadth-First Search (BFS)

Breadth-First Search implements the idea directly. Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier (just like water-front), i.e., the algorithm discovers all vertices in L_k before discovering any vertices L_{k+1}.

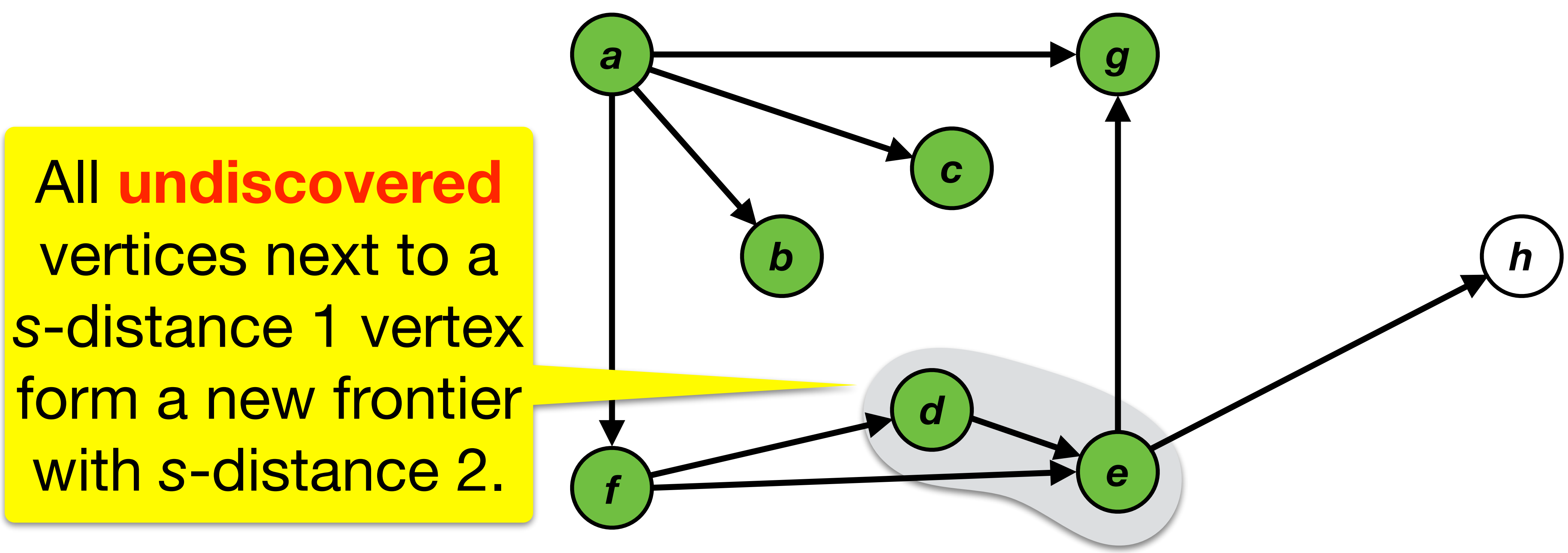
source *s* is vertex *a*



Breadth-First Search (BFS)

Breadth-First Search implements the idea directly. Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier (just like water-front), i.e., the algorithm discovers all vertices in L_k before discovering any vertices L_{k+1}.

source *s* is vertex *a*

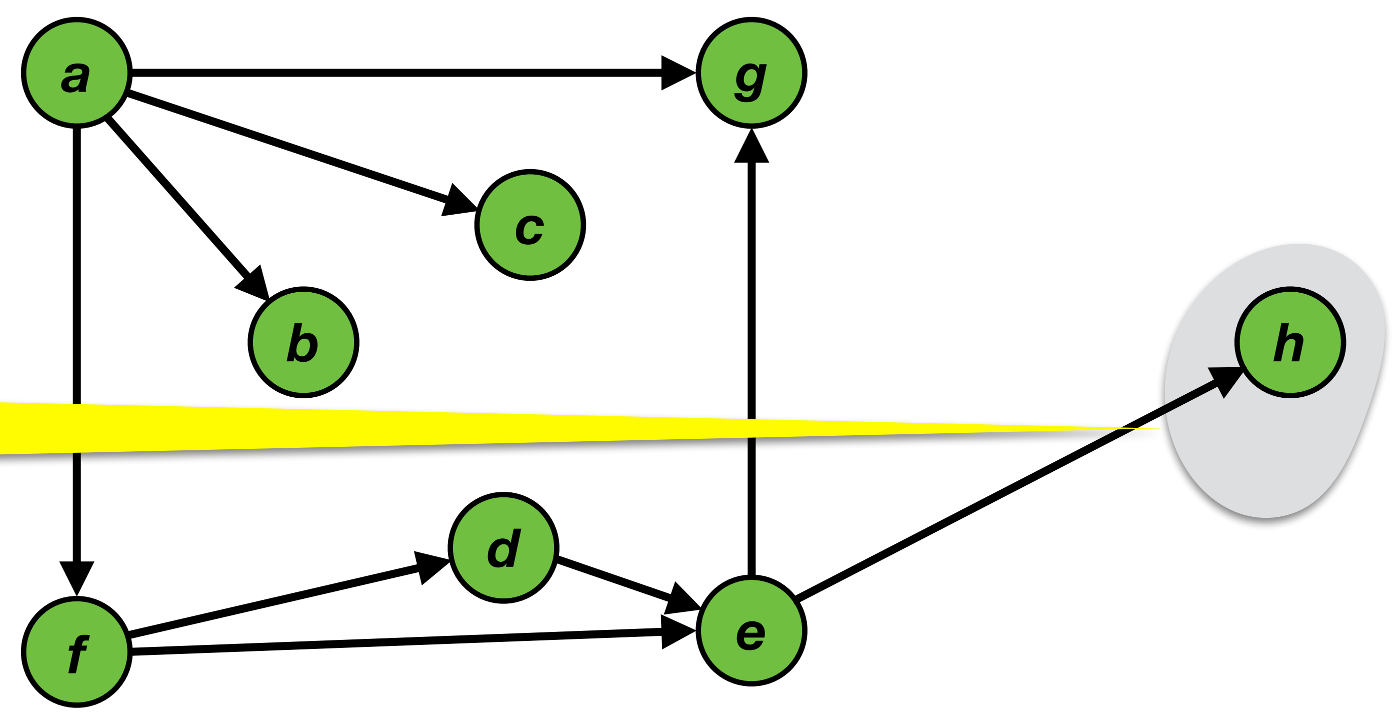


Breadth-First Search (BFS)

Breadth-First Search implements the idea directly. Breadth-first means to expand the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier (just like water-front), i.e., the algorithm discovers all vertices in L_k before discovering any vertices L_{k+1}.

source *s* is vertex *a*

All **undiscovered** vertices next to a *s*-distance 2 vertex form a new frontier with *s*-distance 3.



Breadth-First Search (BFS)

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i+1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
        Add v to the list L[i+1]
      Endif
    Endfor
    Increment the layer counter i by one
  Endwhile
```

Time Complexity of BFS

1) Every vertex will be put in some *L[i]* exactly once and be checked once. This takes $O(|V|)$ *step* .
2) When we explore a vertex, we explore all its adjacency edges once. This takes $O(|E|)$ steps.

In sum, the time complexity of BFS is $O(|V| + |E|)$.

Implementation in the Textbook

```
initialize dist(s) = 0 and dist(u) = ∞ for all other u ∈ V.
initialize queue Q = [s] (a queue containing just s).
while Q is not empty :
    u = eject(Q).
    for all edges (u, v) ∈ E :
        if dist(v) = ∞ :
            inject(Q, v).
            dist(v) = dist(u) + 1.
```

Note: The two implementations are essentially the same.

Exercise: Give an implementation of DFS similar to the above, using a stack instead of a queue.

Retrieving the Shortest Path

In our discussion, we only focused on how to determine dist(u).

Can we also retrieve the shortest path?

- This is easy!
  - For each vertex v, let the algorithm remember *prev[v]*, the vertex immediately precedes *v* in shortest path.
  - To do that, each time that the algorithm discovers a new vertex *v* through an edge *(u, v)*, let *prev[v]* = *u*.

DFS vs. BFS

Why two different search algorithms?

	DFS	BFS
Detecting cycles	✔	✘[1]
Topological ordering	✔	✘[2]
Finding CCs	✔	✔
Finding SCCs	✔	✘[3]
Shortest path problem	✘[4]	✔

1. When BFS encounters a non-tree edge, it is not easy to check whether it is a back edge.
2. The “post-ordering” numbers in BFS are not meaningful.
3. Same as above.
4. DFS focuses on going deep instead of using the shortest path.