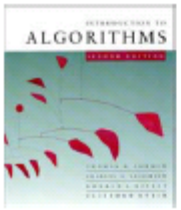
Breadth First Search Algorithm

Instructor: Dr. Ahmed Zekri

The slides are adapted from lectures of Prof. Charles Leiserson , MIT, CLRC textbook 2nd ed. & youtube lecture: Introduction to Graphs(mycodeschool)

Agenda

- Paths in Graphs
- Breadth first search BFS algorithm
- Analysis of BFS time complexity

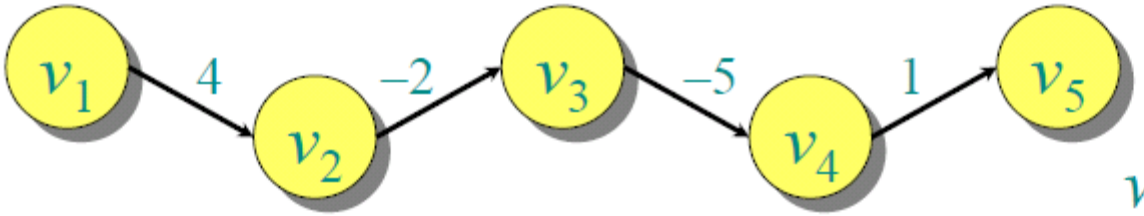


Paths in graphs

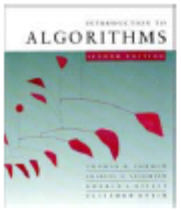
Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The ***weight*** of path $p = v_1 \to v_2 \to \cdots \to v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

Example:



$w(p) = -2$

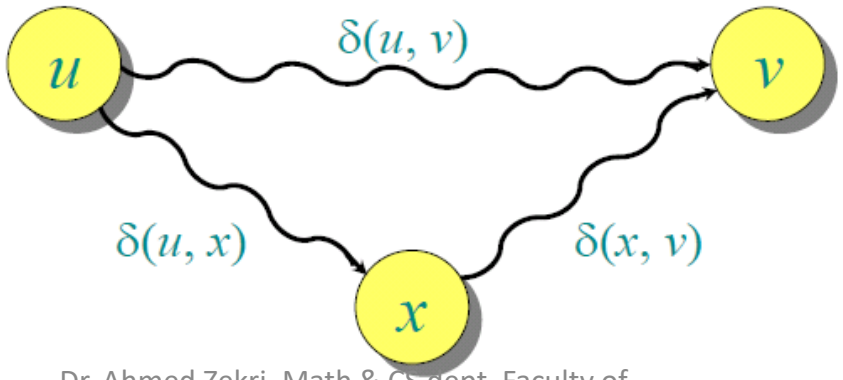


Shortest paths

A ***shortest path*** from u to v is a path of minimum weight from u to v. The ***shortest-path weight*** from u to v is defined as

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

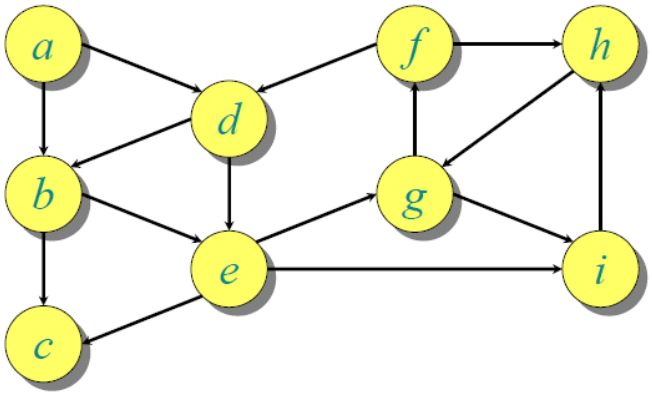
Note: $\delta(u, v) = \infty$ if no path from u to v exists.



Single-source shortest paths

Problem:

- **From a given source vertex *s ∈ V, find the shortest-path weights δ(s, v) for all v ∈ V***
- If all edge weights *w(u, v) are nonnegative, then all shortest-path weights must exist.*

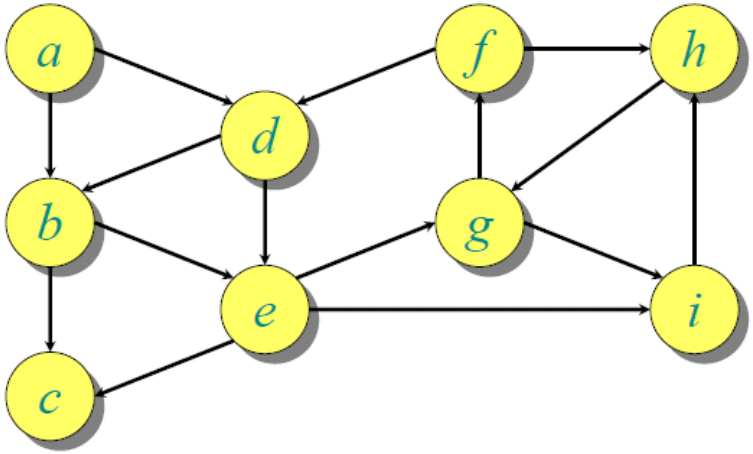


Breadth-first search Algorithm

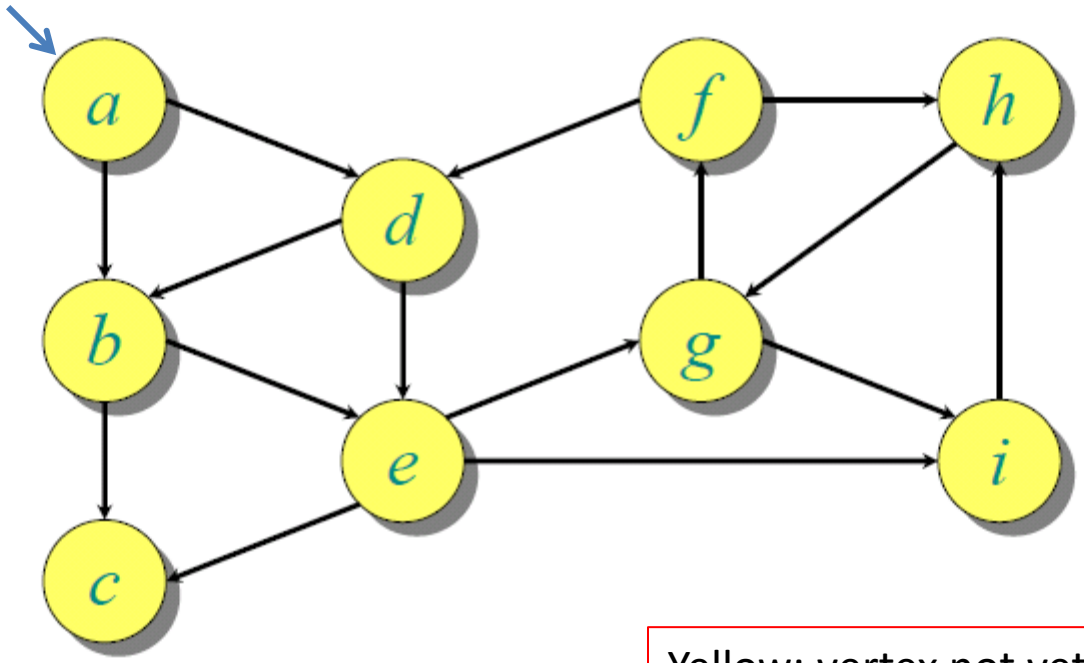
- ***BFS*** is one of the simplest algorithms for searching a graph
- Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, BFS systematically explores the edges of G to "discover" every vertex that is reachable from s.
- BFS computes the **distance** (smallest number of edges) from s to each reachable vertex.
- It also produces a "breadth-first tree" with root ***s*** that contains all reachable vertices from **s**.

BFS …

- For any vertex *v* reachable from *s*, the path in the breadth-first tree from *s* to *v* corresponds to a "shortest path" from *s* to *v* in *G (i.e.,* a path containing the smallest number of edges to go from s to v)
- The algorithm works on both directed and undirected graphs.

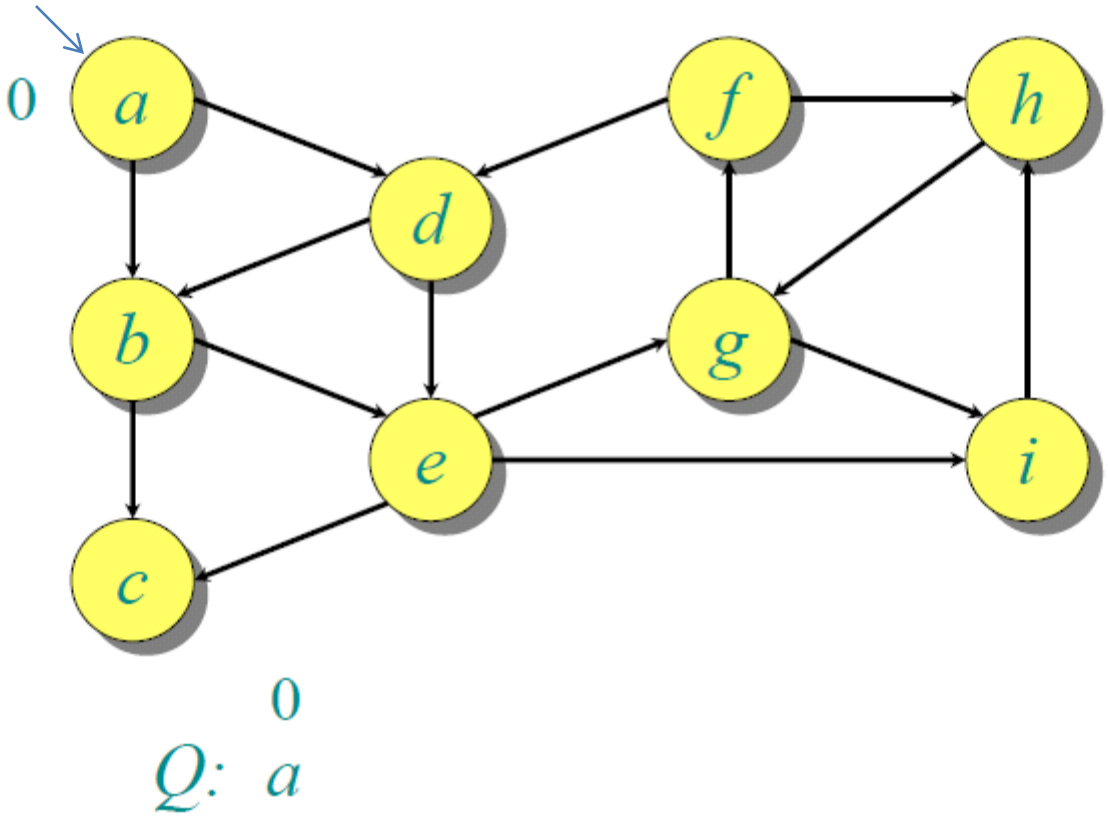


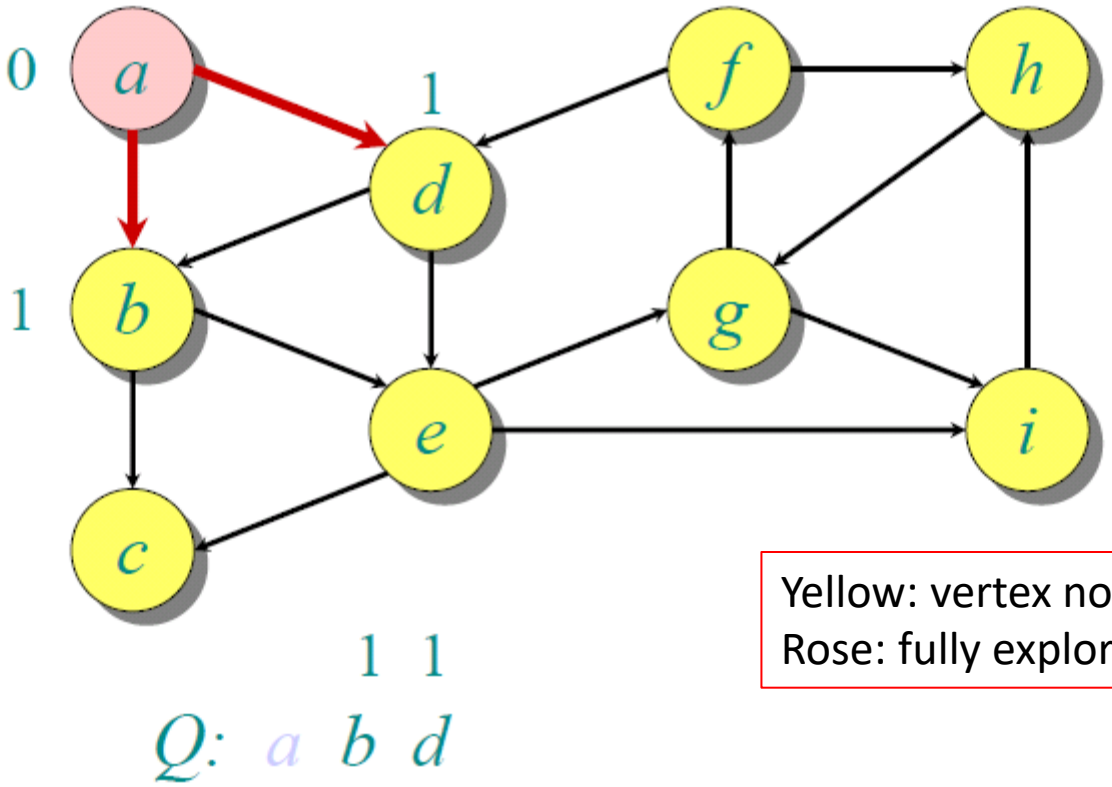
Example 1



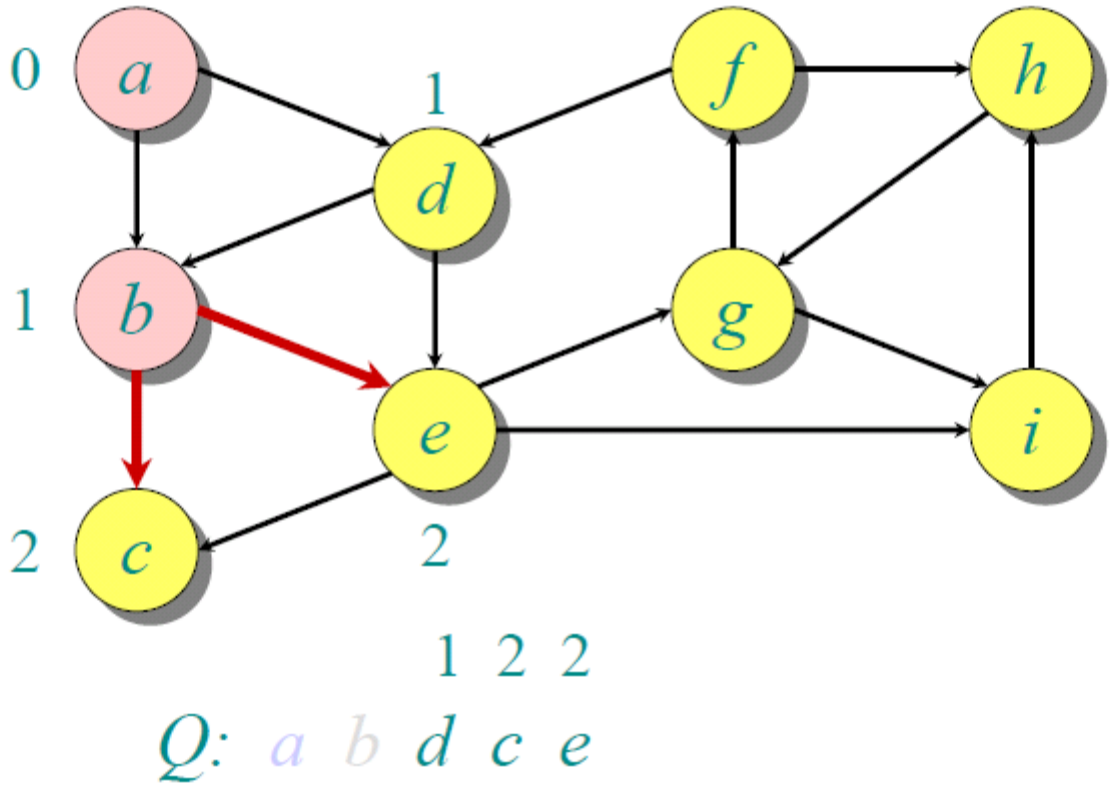
Yellow: vertex not yet visited
Rose: fully explored

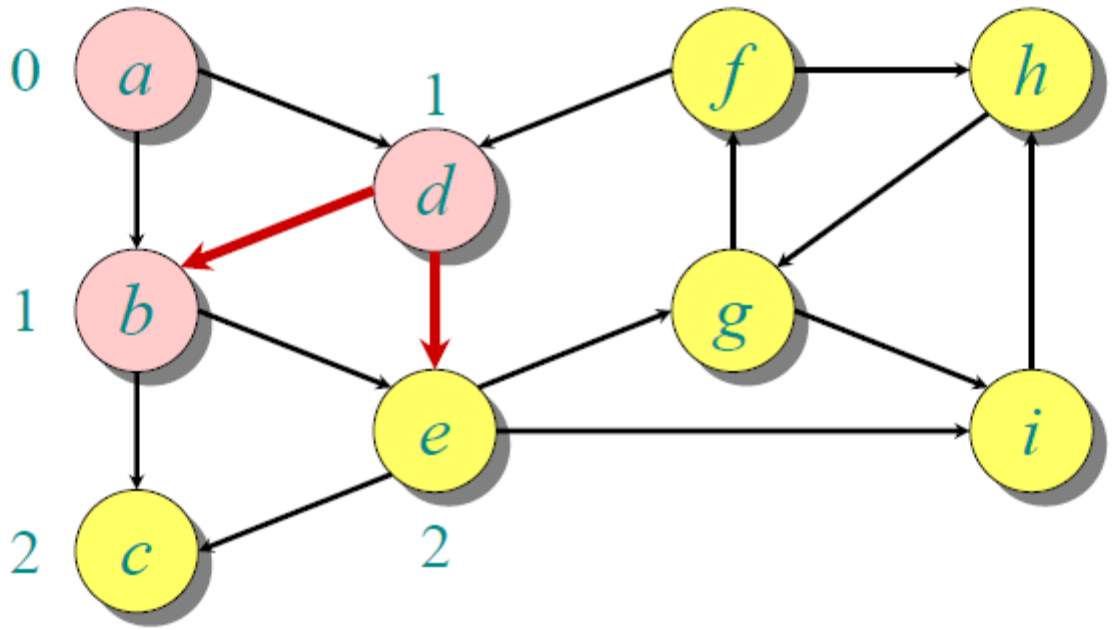
Queue DS Q:

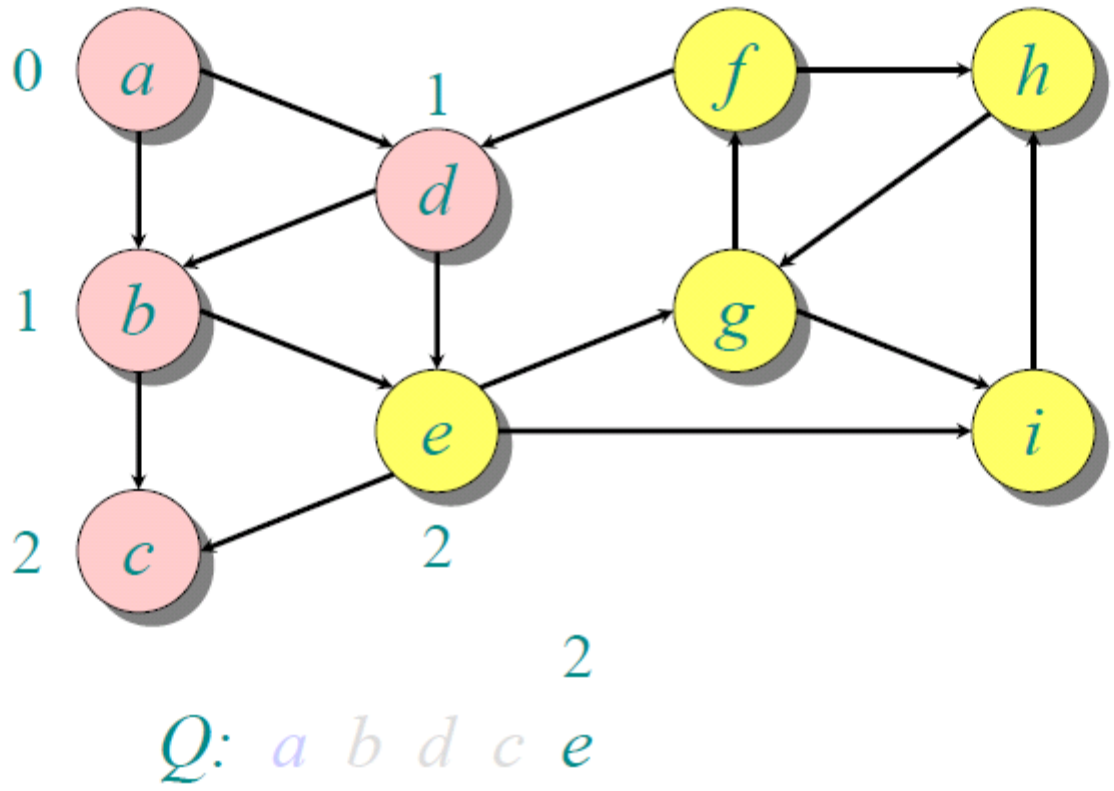

0
a
f
h
d
b
g
e
i
c
0
Q: a

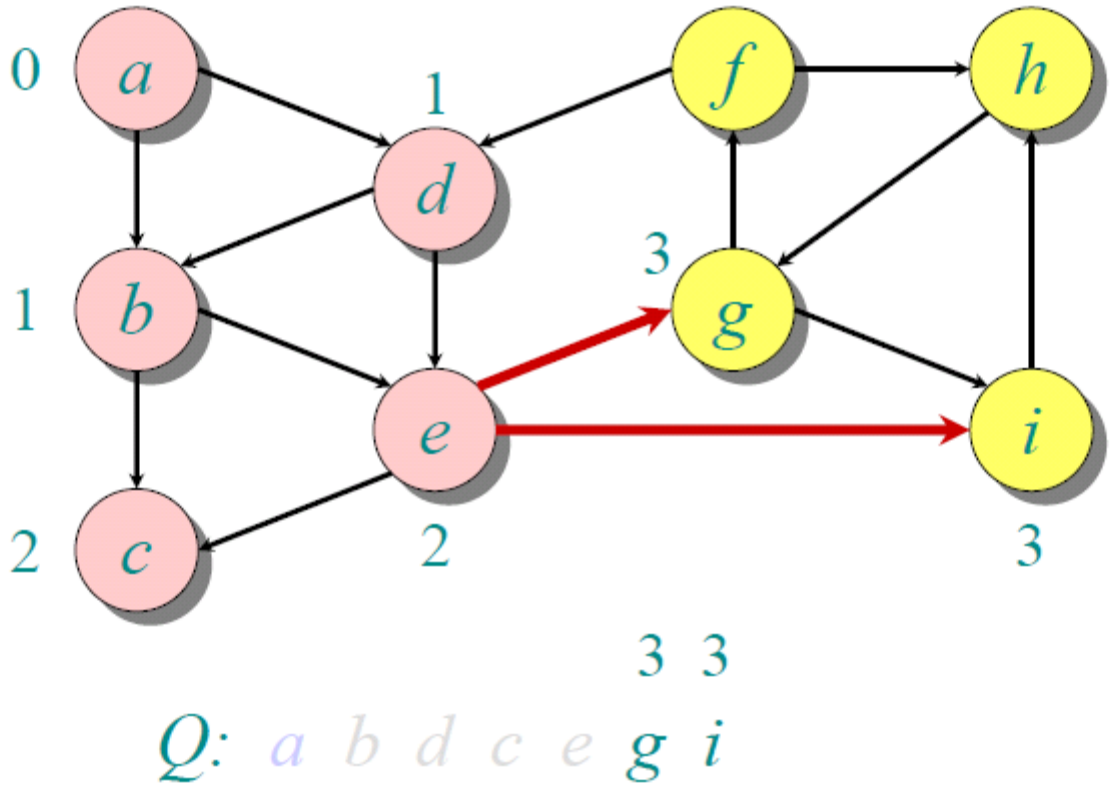


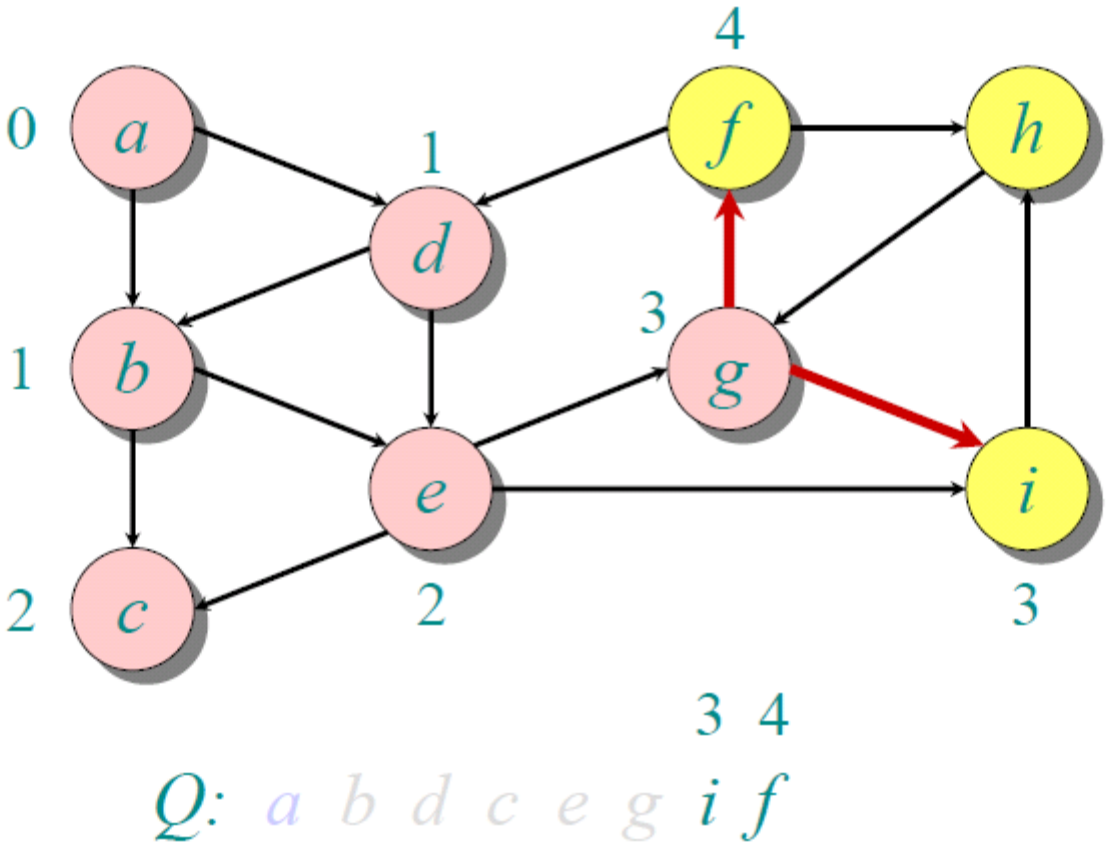
Yellow: vertex not yet visited
Rose: fully explored

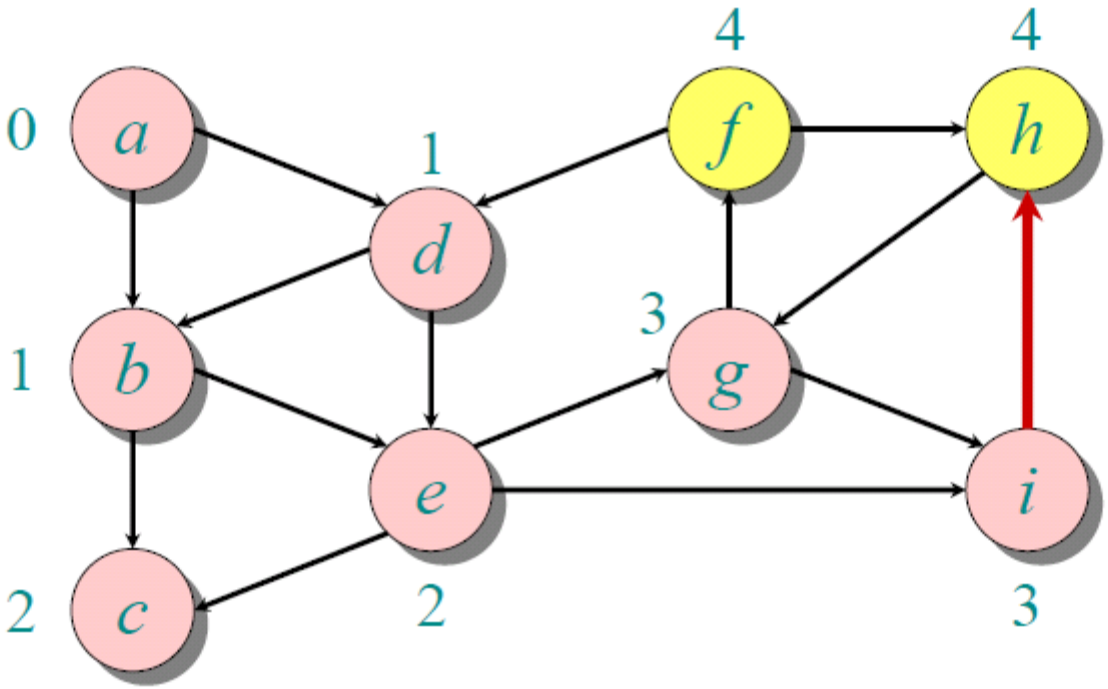

0
a
1
f
h
d
1
b
g
e
i
2
c
2
1 2 2
Q: a b d c e

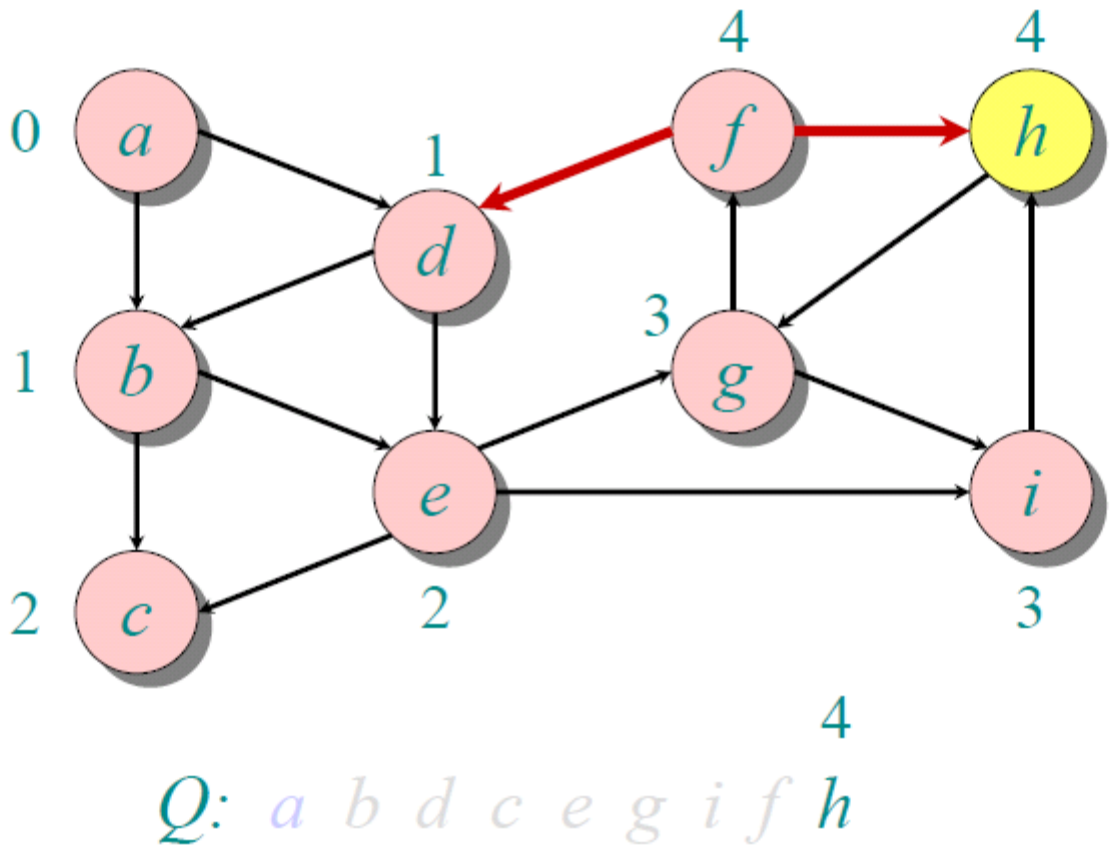

0
a
1
d
f
h
1
b
g
e
i
2
c
2
2 2
Q: a b d c e

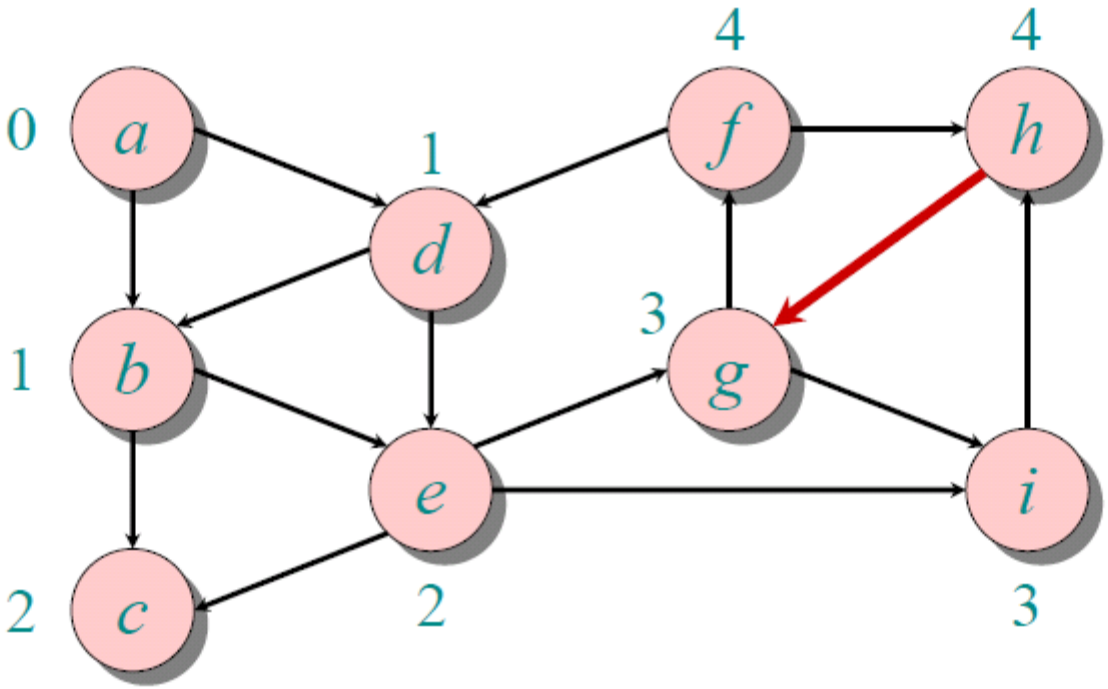

0
a
1
f
h
d
1
b
g
e
i
2
c
2
2
Q: a b d c e


0
a
1
d
f
h
1
b
3
g
e
i
2
c
2
3
3 3
Q: a b d c e g i

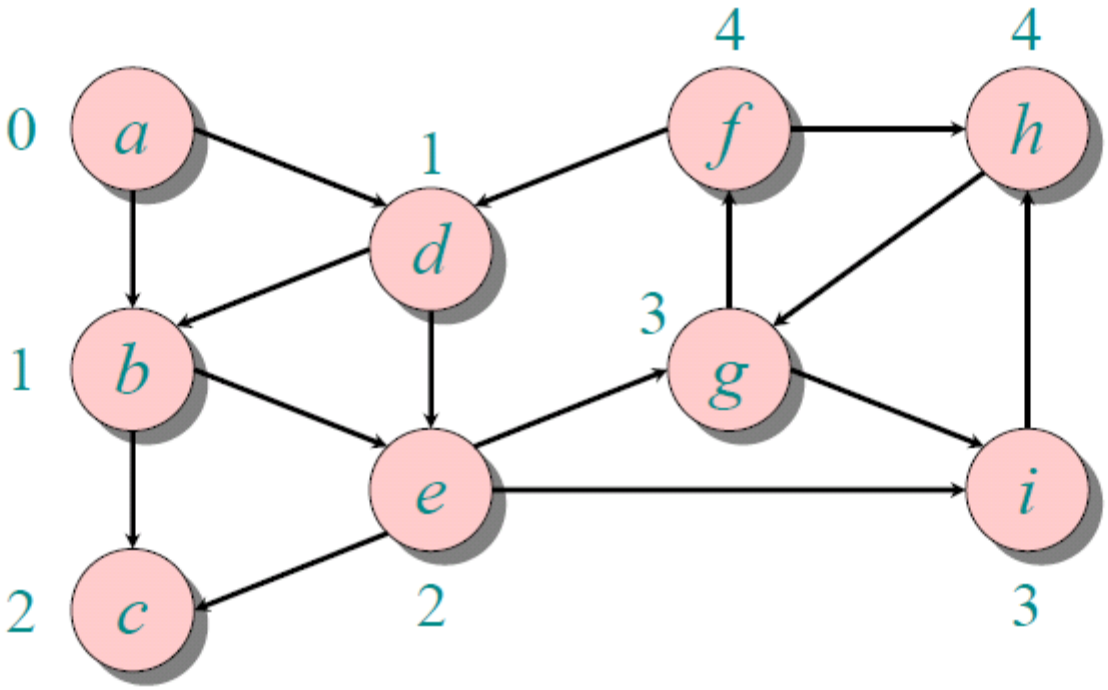

4
0
a
f
h
1
d
3
1
b
g
e
i
2
c
2
3
3 4
Q: a b d c e g i f


4
4
0
a
f
h
1
d
3
1
b
g
e
i
2
c
2
3
4 4
Q: a b d c e g i f h


4
4
0
a
f
h
1
d
3
1
b
g
e
i
2
c
2
3
4
Q: a b d c e g i f h



Q: a b d c e g i f h



Q: a b d c e g i f h

Draw the BFS tree?

BFS implementation

- To facilitate the identification of the vertices, the following colors will be used to describe the state of a vertex:
- White: initial color for all vertices
- Gray: reached (visited)
- Black: fully explored (its adj. list is fully explored)
  - In example1, yellow (white), rose (black)

BFS algorithm

-BFS assumes all w(u, v) = 1

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞        (distance from s to u)
 4          π[u] ← NIL      (The parent of u)
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅                   ( FIFO queue )
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13              do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Aggregate Analysis

- In BFS algorithm, it is not easy to figure out the number of repetition of the loop.
- Aggregate analysis depend on figuring out how many times, in terms of |E|, |G|, the operations (Enque and Deque) in the loop will be executed.

Analysis of BFS

- The overhead for initialization in lines 1-9 is $O(V)$.
  - After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is ***enqueued*** (line 17) at most once (hence ***dequeued*** (line 11) at most once.)
- The operations of **enqueue** and **dequeue** take $O(1)$ time,
  - Overall, the total time in BFS devoted to queue operations is $O(V)$ [aggregate analysis], How?

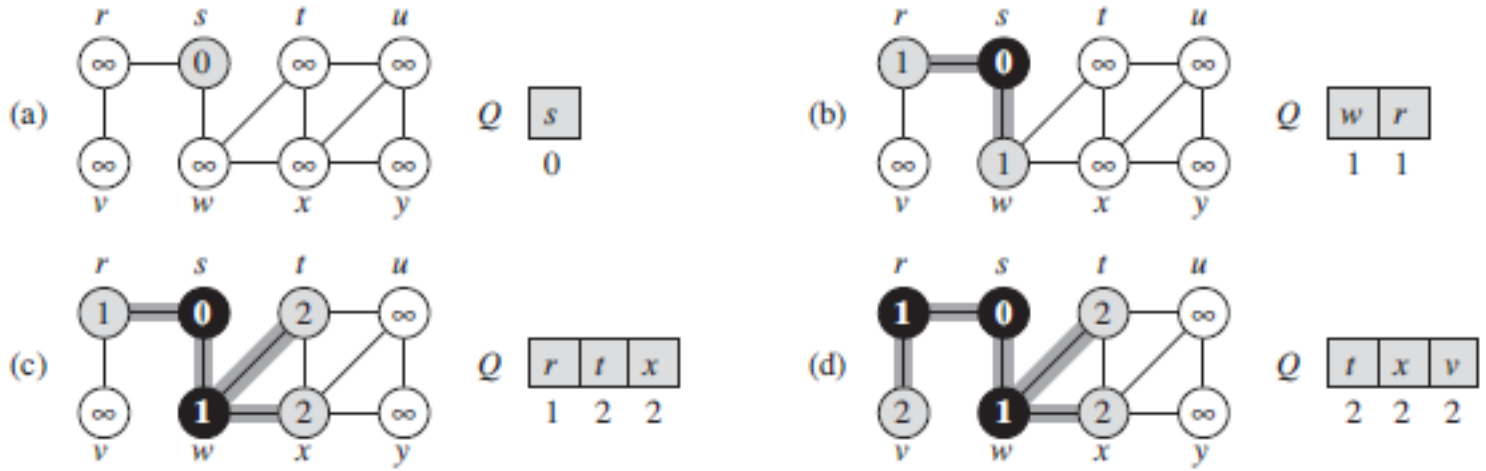
Analysis of BFS

- Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.
- Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.(Lines 12-13)
- **Thus the total running time of BFS is**
- ***O(v) + O(E) = O(V + E).***

Conclusions

- BFS can be used to find the shortest distance between a source vertex and a destination vertex (in terms of the number of edges used to reach the destination)
- BFS applies a greedy strategy since it connects the currently visited vertex with an edge very near to it, the adjacent to it! (greedy)
- The runtime analysis applied in BFS is called aggregate analysis, e.g. finding the overall cost of basic operation by tracing how many times it is applied in terms of V and E.

Example 2



The weights of the edges in BFS are assumed equal to 1.

Figure 22.3: The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the *while* loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

Example 2

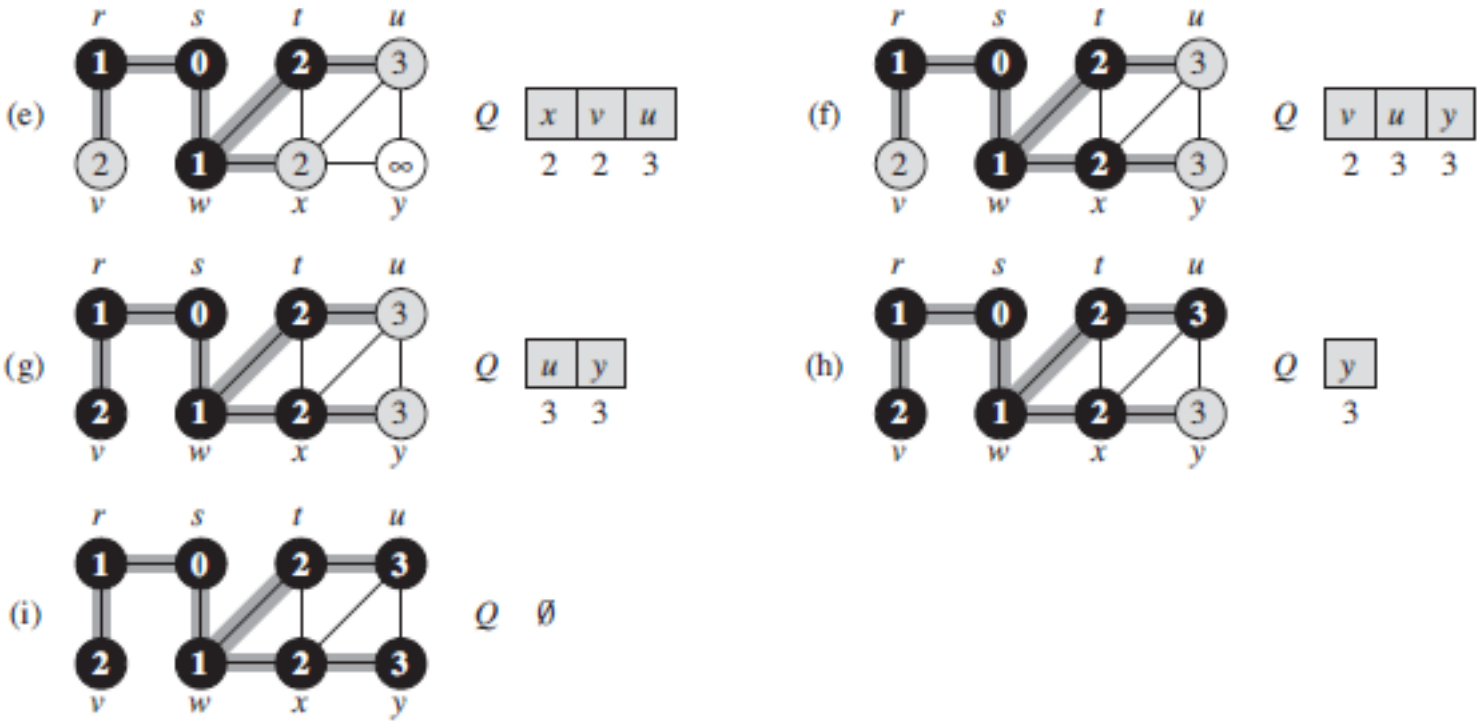
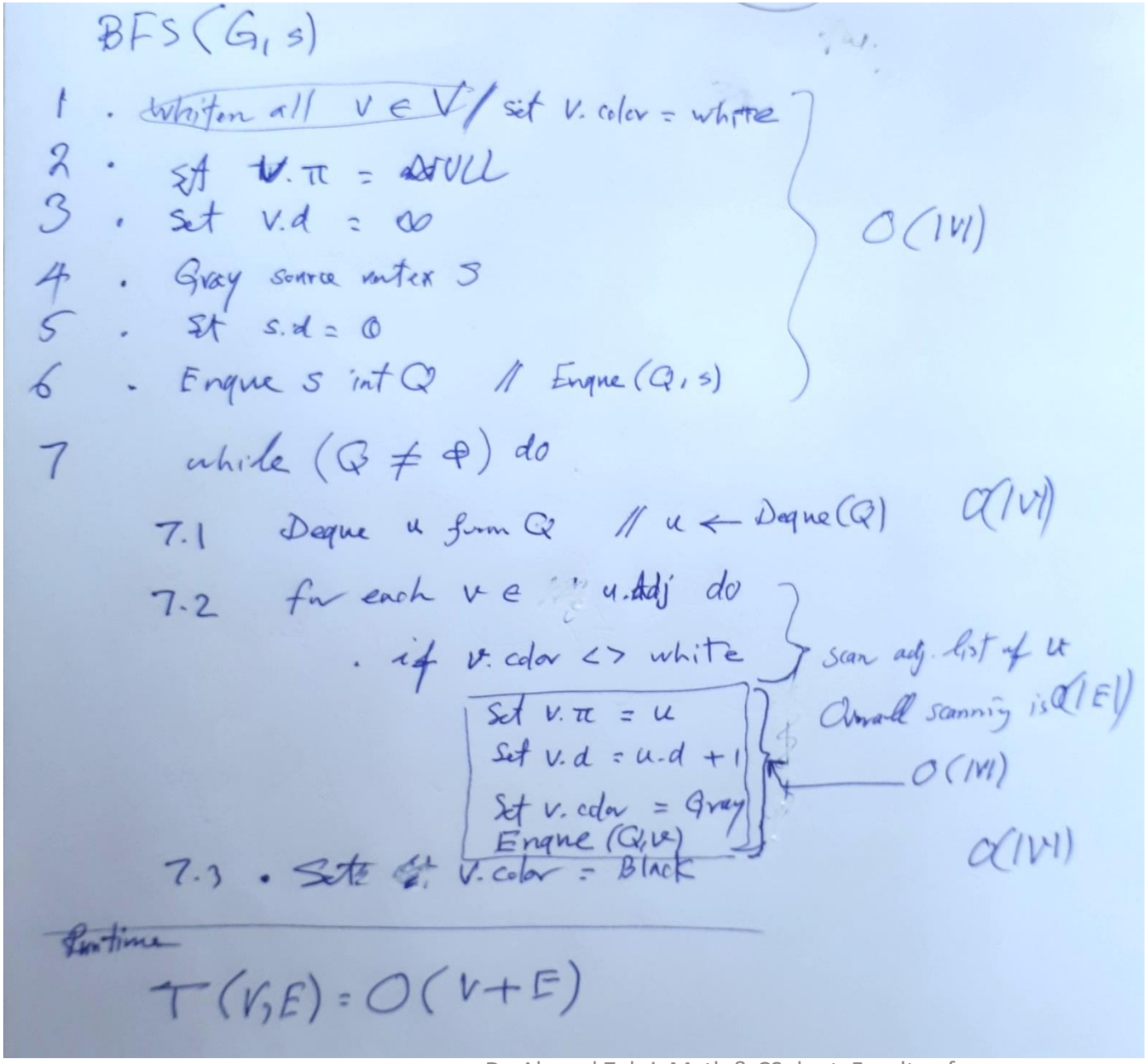


Figure 22.3: The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the *while* loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

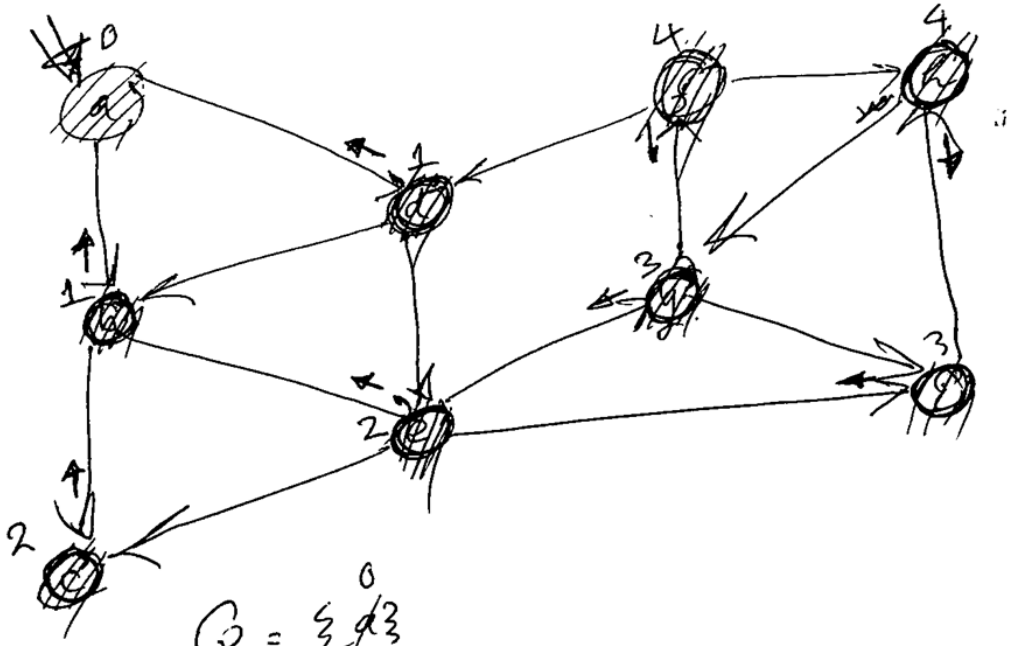
BFS algorithm

```
BFS(G, s)
 1  for each vertex u ∈ V[G] - {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞        (distance from s to u)
 4          π[u] ← NIL      (The parent of u)
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅  ( FIFO queue )
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13              do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```


BFS(G, s)
1. Initialize all v ∈ V // set v.color = white
2. Set v.π = NULL
3. Set v.d = ∞
4. Gray source vertex s
5. Set s.d = 0
6. Enque s into Q // Enque(Q, s)
O(|V|)
7 while (Q ≠ ∅) do
7.1 Deque u from Q // u ← Deque(Q)
O(|V|)
7.2 for each v ∈ u.adj do
. if v.color <> white
scan adj. list of u
Overall scanning is O(|E|)
Set v.π = u
Set v.d = u.d + 1
Set v.color = Gray
Enque(Q, v)
O(|V|)
7.3 . Set v.color = Black
O(|V|)
Runtime
T(V, E) = O(V + E)

Implementation

- Queue data structure is used to keep track of the vertices visited b the algorithm.
- Two operations:
  - *Enque* (insert), *Deque* (delete)
- To analyze BFS, we use aggregate analysis: how many times (in terms of |E| and |V|) the *enque* and *deque* is performed.



$Q = \{s\}$

$= \{b, d\}$

$= \{d, c, e\}$

$= \{c, e\}$ no Enqueue

$= \{e\}$

$= \{g, i\}$

$= \{i, f\}$

$= \{f, h\}$

$= \{h\}$

$= \{\}$

Each vertex is Enqueued at most once in Q

Enqueue → O(|V|)

Dequeue →

- Calculation of distance of each vertex

The algo. pass by each adjacency list at most once

⇒ O(|E|)

The procedure BFS works as follows. Lines 1-4 paint every vertex white, set $d[u]$ to be infinity for each vertex u, and set the parent of every vertex to be NIL.Line 5 paints the source vertex s gray, since it is considered to be discovered when the procedure begins. Line 6 initializes $d[s]$ to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8-9 initialize Q to the queue containing just the vertex s.

The **while** loop of lines 10-18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

- At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12-17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14-17. It is first grayed, and its distance $d[v]$ is set to $d[u]+1$. Then, u is recorded as its parent. Finally, it is placed at the tail of the queue Q. When all the vertices on u's adjacency list have been examined, u is blackened in lines 11-18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).