Graph

- Set of edges (E) and vertices (V).
- Graph applications (Maps, social network, network topologies etc.)
- Directed graph application (FB-friends) and undirected graph application (Instagram followers).
- Adjacency matrix and adjacency list are used to represent the graph.
- Adjacency matrix is preferred for dense and adjacency list is preferred for sparse graph.
- Maximum number of edges in a completely connected graph (where each vertex is directly connected to all other vertices in the graph)
  - In Directed graph: V * (V-1)
  - In Undirected graph: V*(V-1) / 2
- A graph is considered dense if number of edges |E| are closer to V^2 and sparse if number of edges |E| are closer to **V**.
- For dense graph, space complexity of adjacency list is more than the space complexity of adjacency matrix.

Graph Traversal:

Breadth First Search (BFS): Queue data structure is used in BFS.

Attributes of each vertex: (**Color, d, and π**)

Color:

White: (vertex is not visited/discovered). A vertex is said to be discovered when first time it is encountered during the search.
Gray: (vertex is visited/discovered. May have some undiscovered adjacent vertices)
Black: (vertex is visited/discovered and all adjacent vertices are also discovered)

d:

Distance from source vertex to the given vertex. **For a weighted graph** this distance will be the **weight of edges from source vertex** and **for unweighted graph** this distance will be the **number of edges from the source vertex**.

π:

Predecessor/parent vertex.

Main idea:
Given a graph "**G"** and source vertex "**s**". Initially all the vertices in the graph are unvisited so set the attributes of each vertex i.e., (color = white, d = ∞, π = NULL) since we don't know the number of edges from source vertex to all other vertices, so we are initializing them with infinity. Similarly, we don't know the predecessor/parent of each vertex so initialized with NULL.
Now start traversing from source vertex so update the attributes of source vertex i.e., (color = gray, d = 0). Create a queue an insert the source vertex in the queue.
Iterate until the queue is not empty. In every iteration of the loop perform the following steps:

- remove the vertex from the queue (dequeue operation)
- check all the adjacent vertices of the removed vertex and if any of the adjacent vertex is undiscovered/unvisited yet then update the attributes of that undiscovered adjacent vertex i.e., (color = gray, etc.) and then insert that vertex back into the queue.
- Once all the adjacent vertices of the removed vertex are discovered then update the "color" attribute to black.

BFS Algorithm: Given a graph “G” and source vertex “s”.
G.V means vertices of the graph.

```
BFS(G, s)
{
// Initialize all the vertices of graph
  for each vertex "u" belongs to G.V
  {
    u.color = white   (initially all vertices were unvisited)
    u.d = ∞   (initially the distance to reach each vertex is infinity)
    u.π = NULL  (predecessor of each vertex is unknown)
  }
//udate the attricutes of source vertex
  s.color = gray   //source vertex visited
  s.d = 0  // No edge required to reach the source vertex.

  Q = {}  // create an empty queue for BFS
  Enqueue (Q, s)    //insert source vertex in the queue

// main functionality to traverse each vertex in graph
  while (Q ≠ {} )      // iterate until the queue is not empty
  {
    u = deQueue(Q)    // remove the vertex from queue
    for each adj vertex “v” belongs to G.adj[u]     // iterate for all adj vertices of “u”
    {
      If (v.color == white)  //only cater the undiscovered adj vertices and update the attributes.
      {
        v.color = gray  //vertex discovered.
        v.d = u.d + 1  //distance will be #of edges required to reach “u” plus 1 to reach “v”
        v. π = u   //predecessor of “v” will be “u”
        Enqueue (Q, v)   //insert the adj vertex in the queue.
      }
    }
    u.color = black   //since all the adj vertices of “u” are discovered so update the color to black.
  }
}
```

A sample example of BFS is also provided for the reference.

BFS (Sample Example) Distance from source vertex is written inside each node.

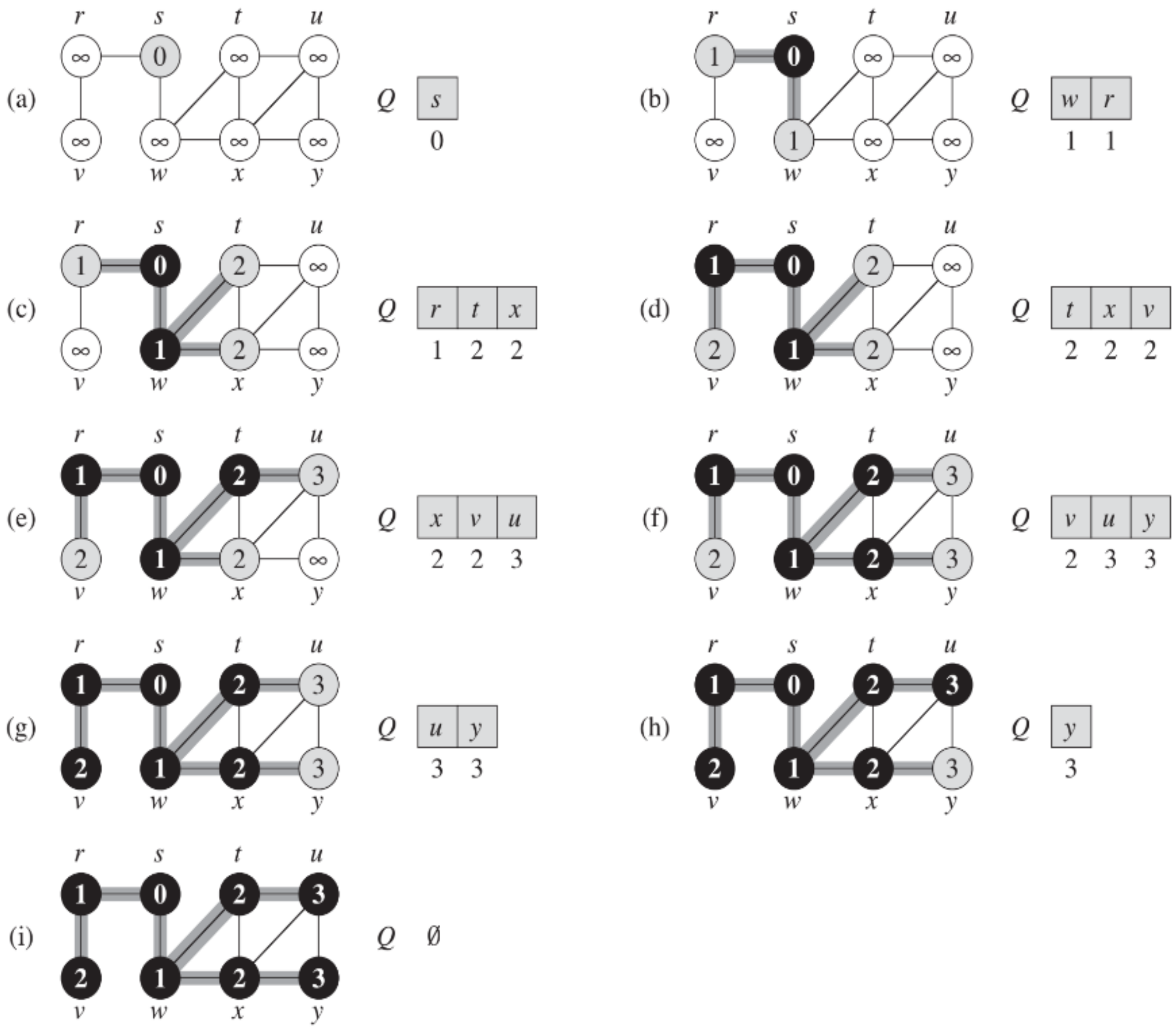

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex u. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear below vertices in the queue.