Breadth-first search

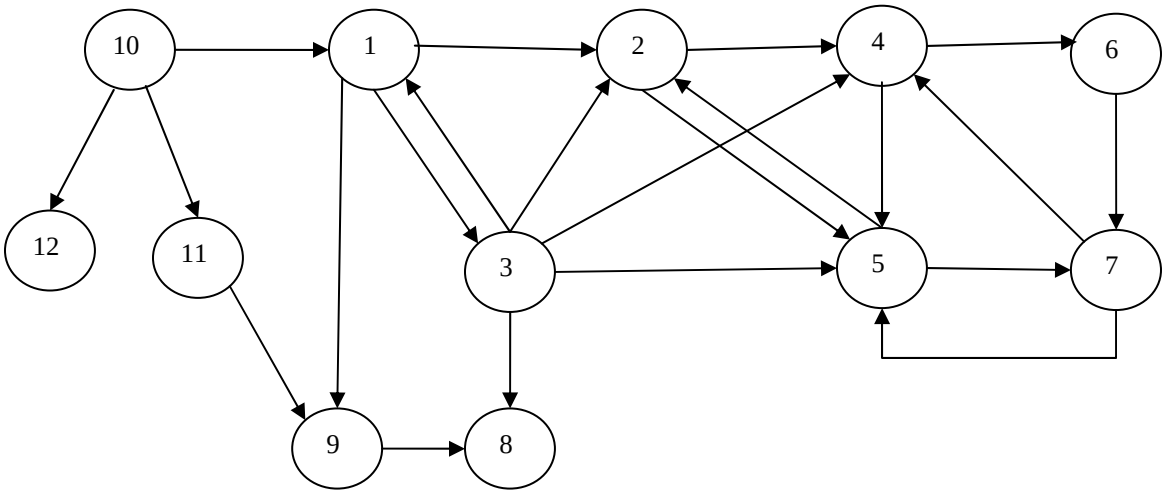
The breadth-first algorithm visits all adjacent nodes (children) before visiting their adjacent nodes, and so on. As with depth-first, when we visit each vertex, we have to decide in what order to visit the adjacent vertex. We will assume an underlying ordering implied by the node number (or index). We start at vertex 1 (or 0 depending on where we start counting) and consider a smaller numbered node before a larger numbered node.

```
void breadthFirstSearch(. . .) {
      // mark all the vertices as not visited

      for v = 1 to n {
            if (v is not visited)
                  bfs(v);
      }
}

void bfs(v ...){
      mark v as visited
      Q.enqueue(v);
      while (Q is not empty) {
         x = Q.dequeue();
         output x (or do whatever, output give breadth-first ordering)
         for each vertex w adjacent to x {
            if (w is not visited) {
                  mark w as visited
                  Q.enqueue(w);
            }
         }
      }
}
```

Use the same example (unconnected graph) as used with depth-first:



The breadth-first ordering is 1 2 3 9 4 5 8 6 7 10 11 12 .

Uses of Breadth-first search

Breadth-first Traversing is often used for the **processing of digital images**. It presents efficient algorithms for eroding, dilating, skeletonizing, and distance-transforming regions. These algorithms work by traversing regions in a breadth-first manner, using a queue for storage of unprocessed pixels. They use memory efficiently — pixels are removed from the queue as soon as their processing has been completed — and they process only pixels in the region (and their neighbors), rather than requiring a complete scan of the image. The image is still represented as a pixel matrix in memory; the graph is just a convenient framework for thinking about the algorithms.

Breadth-first search (BFS) is used for finding all connected components in a graph (finding all nodes within one connected component). The set of nodes reached by a BFS are the largest connected component containing the start node.

A variation of BFS is used for finding the shortest path between two nodes u and v (in an unweighted graph). It is also used for finding the shortest path between two nodes u and v (in a weighted graph). This is Dijkstra's algorithm.

BFS can be used to test bipartiteness (when a graph's set of vertices can be divided into two disjoint sets so that every edge in the graph connects a vertex from one set to the other, see Section 9.2 in the text). The algorithm starts the search at any vertex and gives alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbors, 0 to those neighbors' neighbors, and so on. If at any step a vertex has (visited) neighbors with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

Uses of Depth-first search

A **biconnected graph** is a graph where the removal of no vertices disconnects the graph. This is important with networks. If one computer goes down, the network should not be affected. Similarly in a mass transit system, there needs to be alternate routes. An **articulation point** is a node whose removal would disconnect the graph (critical to problems mentioned). All articulation points in a connected graph can be found using depth-first search.

An **Euler circuit** is a path that visits every edge exactly once. (In 1736, Euler proposed the problem of finding these circuits. This was the beginning of graph theory.) This problem is solved using a variation of depth-first search: keep removing components that get you to where you started from.

Other questions that can be answered:

- Is a graph connected? If not, what are its connected components?
- Does a graph have a cycle?
- In a directed graph, is there a path from x to y ?
- Find the transitive closure (add edges so that you can get from any node x to any node y).