CS 228: Introduction to Data Structures
Lecture 37

[redacted]

Properties of Breadth-First Search

The following theorem[1] states the key of BFS.

> **Theorem.** During its execution, BFS(G,s) discovers every node of G that is reachable from s. Upon termination, dist(v) is the length of the shortest path from s to v, for every node v. Further, for any node $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to pred(v), followed by the edge (pred(v),v).

The theorem implies that any node v that has $dist(v) = \infty$ at the end of the execution of BFS(G,s) must be unreachable from s. It also implies that for any node $v \neq s$, the path from s to v in the BFS tree is a shortest path from s to v in the graph.

We can get some intuition as to why BFS correctly computes the distances from s by seeing what happens in the first few iterations. First, note that s is correctly assigned distance 0, and is enqueued first. When s is

[1] You can find a proof in Cormen, Leiserson, Rivest, and Stein's textbook, *Introduction to Algorithms*.

processed, its neighbors, which are all the nodes at a distance of 1 from s, are enqueued, but no other node is. Since we're using a queue, all these nodes will be processed before any nodes enqueued later. As the distance-1 nodes are processed, all the distance-2 nodes are enqueued, because every distance-2 node must be reachable by a single edge from some distance-1 node. No other nodes are enqueued, because

- every node at a distance less than 2 from s is either grey (still in the queue) or black (already processed), and

- no white node v at a distance 3 or greater is reachable by a single edge from a node at a distance of 1 — otherwise v would be at distance 2.

This reasoning can be generalized to all distance values using "mathematical induction". You'll learn about induction in your discrete math class (ComS 330 or CprE 310).

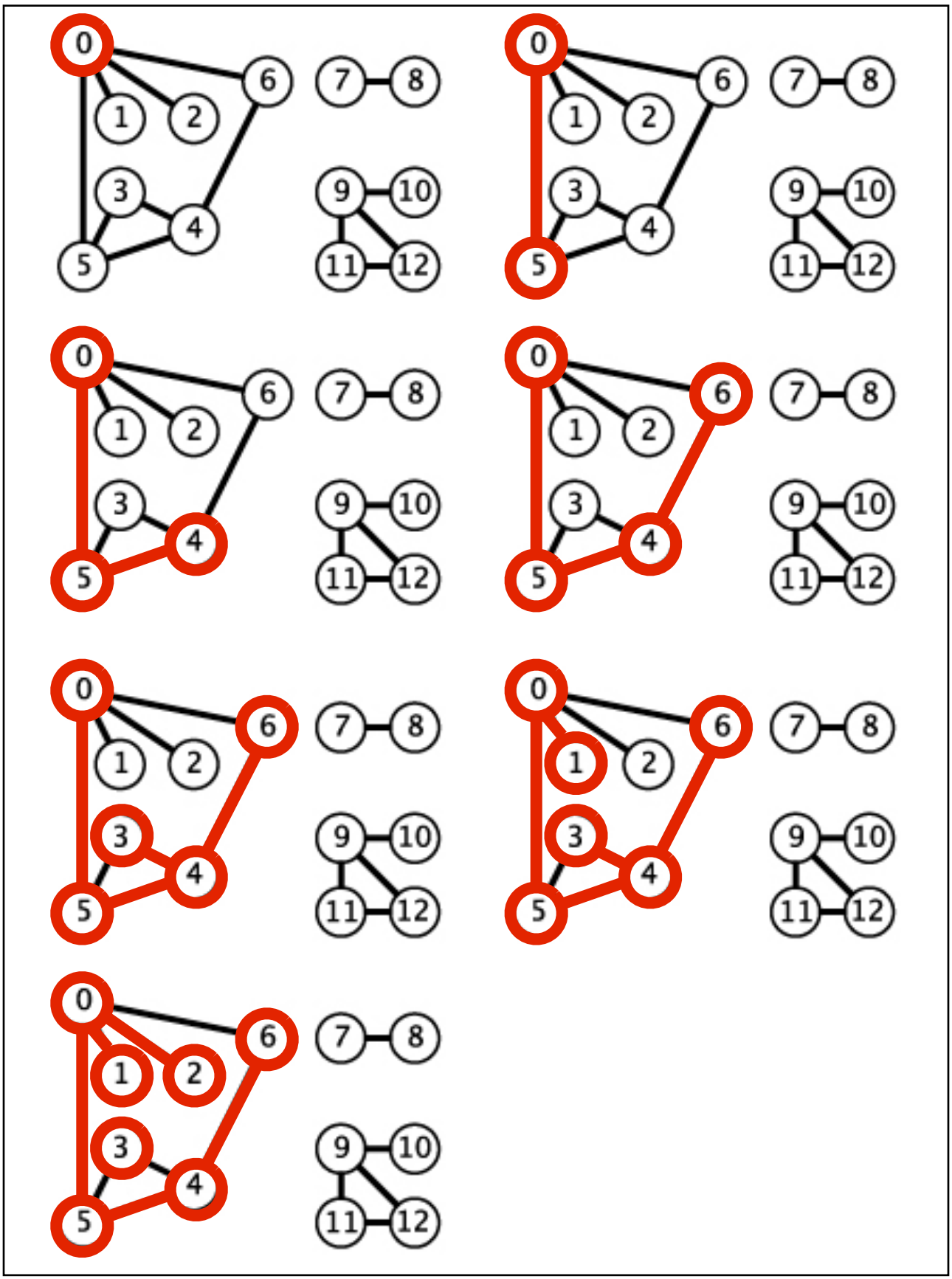
Time complexity. Each node is enqueued at most once and dequeued at most once, so the time spent in queue operations is O(V). The set of neighbors of any vertex is scanned at most once, and the total size of the neighbor sets is O(E). On an adjacency list or a `HashMap`/`HashSet` implementation, scanning the neighbor set takes O(1) time per neighbor, so BFS runs in O(V + E) time. On

an adjacency matrix implementation, scanning the set of neighbors of a node takes O(V) time, so the running is $O(V^2)$.

Depth-First Search

DFS starts at an arbitrary node s and searches a graph as "deeply" as possible as early as possible. For example, if your graph is a tree rooted at s, DFS performs a preorder (or if you prefer, postorder) traversal.

When DFS reaches a node u, it processes u by inspecting each neighbor v of u, v. If v has not yet been seen yet, DFS visits it recursively. If we cannot get anywhere new from u, just backtrack to u's predecessor and resume the search. The process is illustrated in the next figures.


0 1 2 3 4 5 6 7 8 9 10 11 12

If DFS is applied to a connected undirected graph or a strongly connected directed graph, every node of the graph will be visited. In general, however, a DFS started at a node s only visits those nodes that are reachable via a path from s. If we want to explore the entire graph, we have to perform DFS multiple times, with different starting vertices. As in BFS, to prevent the algorithm from visiting any node twice, each node v has a **color**, with the following interpretation.

- color(v) == white means v is undiscovered and unprocessed. Initially, every node is white.

- color(v) == grey means v has been discovered, but has not been processed.

- color(v) == black means v has been discovered and processed.

After a node v has been colored black, we say that v is ***finished***.

The pseudocode consists of two methods, `dfs()` and `dfsVisit()`. `dfs()` initializes color and pred and calls `dfsVisit()` as many times as necessary, until all nodes have been visited. `dfsVisit()` performs the depth-first search, starting at a given node u. As we saw, this only guarantees that we'll visit every node that is reachable

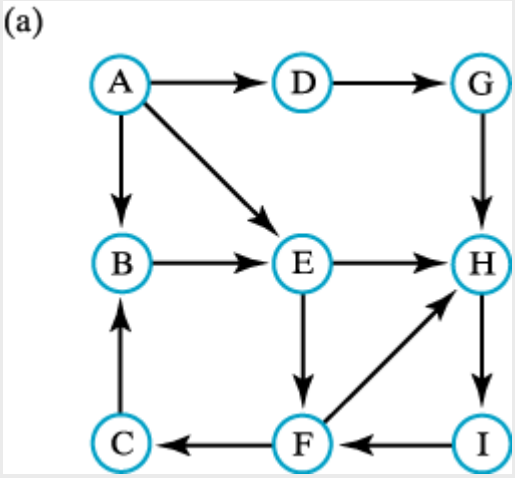
from u. To ensure that all nodes are visited, `dfs()` calls `dfsVisit()` on every white node it encounters.

As in BFS, for every node v, DFS maintains a reference `pred(v)` to its *predecessor* in the search; i.e., the node u that was being processed when v was first discovered. Thus, v was white when u was colored grey. When `dfs()` terminates, `pred()` defines a collection of trees, called a ***DFS forest***. In an undirected graph, each tree in the forest corresponds to a connected component.

```
dfs(G):

   foreach node v in G
      color(v) = white
      pred(v) = null

   foreach node v in G
      if color(v) == white
         dfsVisit(G, v, color, pred)

   return pred
```

```
dfsVisit(G, u, color, pred):
    color(u) = grey
    foreach neighbor v of u
        if color(v) == white
            pred(v) = u
            dfsVisit(G, v, color, pred)
    color(u) = black
```

Example. Suppose we execute DFS on the next graph, starting at A.



Assuming that the neighbors of a node are examined in alphabetical order, dfsVisit(G,A,color,pred) would visit the nodes in the order A, B, E, F, C, H, I, D, G. Here is the

sequence of recursive calls (we omit all arguments to dfsVisit except the node at which the search starts):

```
dfsVisit(A)
 |dfsVisit(B)
 | |dfsVisit(E)
 | | |dfsVisit(F)
 | | | |dfsVisit(C)
 | | | |dfsVisit(H)
 | | | | |dfsVisit(I)
 |dfsVisit(D)
 | |dfsVisit(G)
```

Notice how the method call stack always reflects the path to the current node in the DFS.

Here is the predecessor table:

v	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
pred(v)	null	A	F	A	B	E	D	F	H

Whenever a node is visited during DFS, we usually perform some action on the node. For instance, if we want to count the total population of a graph representing

cities (nodes) connected by roads (edges), we might add the population of the visited city to the grand total.

Time complexity. Let V and E denote the number of vertices and edges in a graph, respectively. Then, DFS runs in O(V + E) time if we use an adjacency list or the `HashMap`/`HashSet` implementation; $O(V^2)$ time if we use an adjacency matrix. Hence, an adjacency list is asymptotically faster if the graph is sparse.

Reachability and the White-Path Theorem

A basic property of DFS is that an invocation of `dfsVist(u)` will visit every unvisited node v that is reachable from u, and that the path it follows to go from u to v consists of only white nodes. More formally, we have the following[2].

> **White-Path Theorem.** In a DFS forest of a (directed or undirected) graph G, node v is a descendant of node u if and only if at the time that the search discovers u, there is a path from u to v consisting entirely of white nodes.

[2] For a proof, see CLRS.

Applications of DFS

DFS has several applications. For instance, we can use it to determine whether there is a path from a node u to another node v. Just do DFS from u: if v gets visited, there is a path; otherwise, there is no path.

Another simple application is finding connected components in an undirected graph G. As we have seen, the DFS forest of G contains as many trees as G has connected components. It is not too hard to modify DFS so that it assigns to each vertex v an integer label `cc(v)` between `0` and `k-1`, where k is the number of connected components of G, such that `cc(u) == cc(v)` if and only if u and v are in the same connected component.

We now discuss two more advanced applications: detecting cycles and topological sort. First, however, we need to analyze the types of edges that are encountered during DFS.

Classification of Edges

We can define four edge types in terms of the DFS forest of a graph G:

1. ***Tree edges*** are edges in the DFS forest. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).

2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a DFS tree. We consider self-loops, which may occur in directed graphs, to be back edges.

3. ***Forward edges*** are those non-tree edges (u, v) connecting a vertex u to a descendant in a DFS tree.

4. ***Cross edges*** are all other edges. They can go between vertices in the same DFS tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different DFS trees.

The figure on the next page[3] illustrates the four edge types. When we first explore an edge (u, v) during a DFS, the color of vertex v tells us something about the edge:

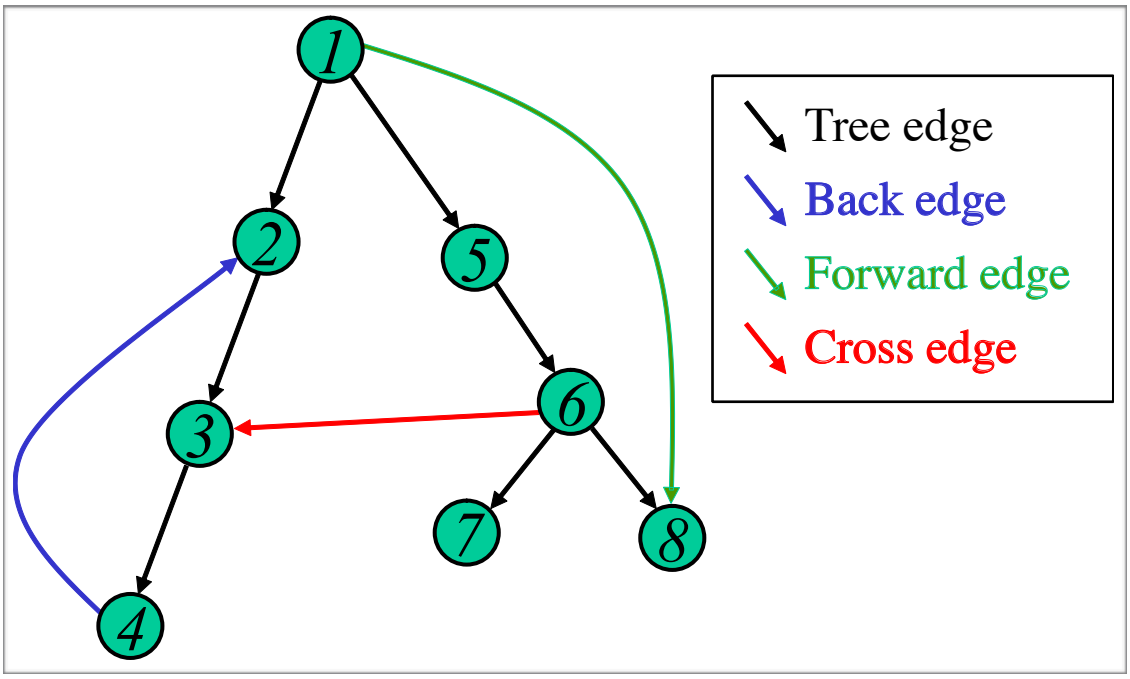
1. WHITE indicates (u, v) is a tree edge,

2. GRAY indicates (u, v) is a back edge, and

3. BLACK indicates (u, v) is a forward or cross edge.

---

[3] http://en.wikipedia.org/wiki/Depth-first_search



The first case is immediate from the specification of DFS. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active `dfsVisit` invocations. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex has reached an ancestor. The third case handles the remaining possibility. There is an efficient way to distinguish between forward and cross edges in this case, but we will not discuss it here (see CLRS for details).

It is clear from the above picture that if we encounter a back edge during the DFS of a directed graph G, then G must have a cycle. We will explore this idea further on Monday.