CS 5084 - Algorithms: Design and Analysis

Section 3.2 Graph Connectivity and Graph Traversal

Joe Johnson

Table of Contents

Introduction

Now that we have some fundamental notions regarding graphs, we turn next to a basic algorithmic question: node-to-node connectivity.

Suppose we are given a graph $G = (V, E)$ and two particular nodes, s and t. We'd like to find an efficient algorithm that answers the following question:

Is there a path from s to t in G? This is typically called the problem of determining *s-t connectivity*.

Introduction - contd.

There are two fundamental approaches to this problem:

- breadth-first search (BFS)
- depth-first search (DFS)

Table of Contents

Breadth First Search

Breadth-first search (BFS) is the simplest algorithm for determining *s-t connectivity* based on the following idea:

In BFS, we search outward from our starting node, s, in all possible directions, adding nodes one layer at a time.

General BFS Algorithm:

1. Start with node s.
2. Visit each node v for which there exists an edge from s to v: *first layer*.
3. Visit each node w for which there exists an edge from a node in the first layer to w: *second layer*.
4. Repeat in this way until there are no more new, connected nodes.

Table of Contents

BFS - Example

Consider the graph in diagram below:

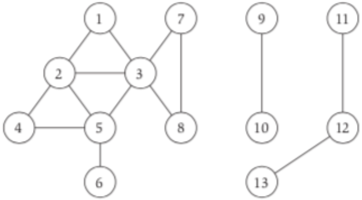


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

Let's run the BFS algorithm on this graph, starting with node, 1.

BFS - Example - contd.

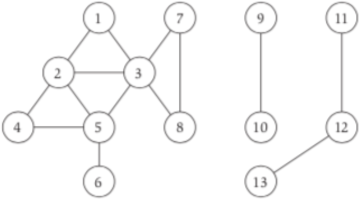


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

1. Start with node 1 as our starting node.
2. First layer: 2 and 3
3. Second layer: 4, 5, 7, 8
4. Third layer: 6
5. Terminate - no more new, connected nodes.

Table of Contents

Layers in BFS

We define the layers, $L_1, L_2, L_3, \ldots$ rigorously as follows:

- Layer L_0 consists of the starting node, s.
- Layer L_1 consists of all nodes that are neighbors of starting node, s.
- Assuming we have defined layers $L_1, \ldots, L_j$ then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.

Layers in BFS - contd.

(3.3) For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There exists a path from node s to node t if and only if t appears in some layer.

Recall our definition of *distance* between nodes u and v which simply refers to the minimum number of edges (hops) on a path between u and v.

Table of Contents

BFS Tree

BFS produces a tree T rooted at the starting node, s on the set of nodes reachable from s.

Consider the diagram below showing the construction of the BFS Tree for the example graph shown earlier:

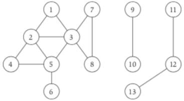
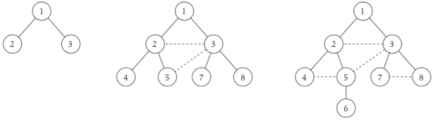


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.



(a) (b) (c)

BFS Tree Theorem (3.4)

(3.4) Let T be a BFS Tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge e in graph, G. Then i and j differ by at most 1.

Proof: by Contradiction. Suppose i and j differ by more than 1, and in particular, $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to x are examined. Since x belongs to layer L_i, the only nodes discovered (or visited) by x are those nodes belonging to layers L_{i+1} and earlier. Since y is a neighbor of x, then y will be discovered by the time we get to layer L_{i+1} at the latest. But if $j = i + 1$, then that would imply that $i = j - 1$ (and thus, $i \not< j - 1$). Thus, we have a contradiction. Thus, i and j differ by at most 1.

Exploring a Connected Component

Definition: Consider the set of nodes that are reachable from starting node s. We call this set, R, the *connected component* of G containing s.

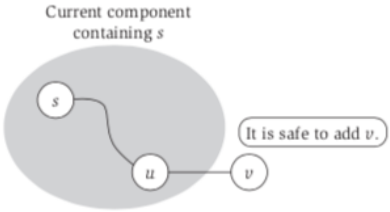
Note that BFS is just one approach that provides for a specific order in which we discover the nodes in R.

Also note that once we have the *connected component* for our starting node, s, we can simply find whether $t \in R$ in order to determine whether there exists a path from s to t.

At a more general level, we can build up the connected component, R, for some starting node s by "exploring" G in *any* order, starting from s.

- To start off we define $R = \{s\}$.
- Then, at any point in time if we find an edge (u, v) where $u \in R$ and $v \notin R$ we can add v to R.

Consider the diagram below:



Consider the following *general, nondeterministic* algorithm for finding the connected component, R:

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

3.5 - The Algorithm for Building Connected Component, R

(3.5) The set R produced by this algorithm is precisely the connected component of G containing s.

Proof: We will show that both of the following are true:

1. For any node, v added to R by the algorithm, there exists a path from s to v.
2. For any node, w *not* added to R by the algorithm, there does *not* exist a path from s to w.

3.5 - Proof - contd.

1. Consider a situation in which there exists a path from s to u, and that there exists an edge from (u, v). Thus, the algorithm will detect this edge, and add v to R. Does there exist a path from s to v, in this situation? Yes - namely, the path from s to u plus the edge from u to v.

3.5 - Proof - contd.

2. by Contradiction. Suppose we have a node $w \notin R$ but that there *does* exist a path, P from s to w. Since $s \in R$ but $w \notin R$, there must be a first node v on P that does not belong to R. Assume this node is some node, v, where v is not equal to s. Thus, there is a node, u immediately preceding v on P, so (u, v) is an edge. Moreover, since v is the first node on P that does not belong to R, we must have $u \in R$. It follows that (u, v) is an edge where $u \in R$ and $v \notin R$. But this contradicts the stopping rule of the algorithm which will add any node v to R where there exists a node $u \in R$, and an edge (u, v). In other words, the algorithm *will* add v to R, contrary to the assumption that $v \notin R$. Thus, if $w \notin R$, then, there *must* not exist a path, P, from s to w.

Table of Contents

DFS Description

Consider a graph G as representing a maze of rooms, where each node represented a particular room and each edge represented a hallway leading from one room to another.

If you were wandering around in this maze, looking for a way to escape, you might start at a particular room, s, and try the first edge leading out of it, to a node, v. Then, you follow the first edge out of v and continue in this way until you reach a "dead end" - i.e., a node for which you had already explored all of its neighbors. At this point, you'd backtrack until you reached a node with an unexplored neighbor, and resume from there.

DFS Description - contd.

This approach to searching a graph is called the depth-first search algorithm (DFS).

Note that DFS, like BFS, is a particular implementation of the connected component algorithm for G containing s.

Recall that our original, motivating problem was to solve the s-t *connectivity problem*. We can apply DFS(s), and find whether node t is in the *connectivity component* of G containing s using this algorithm.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Table of Contents

DFS Tree

Similar to the BFS algorithm, the DFS algorithm also produces a tree, T on the connectivity component of G containing s, rooted at s. However, it works a little differently:

- For a given node, u, we make u the parent of node v if u is responsible for discovering v. That is, if DFS(u) results in a call to DFS(v), then we create an edge in the DFS tree, T, from u to v, (u, v).

DFS Tree - contd.

Recall our example graph, G:

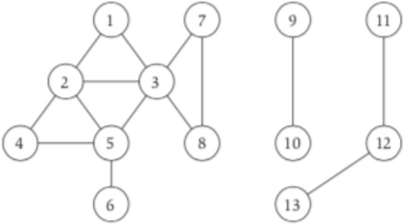
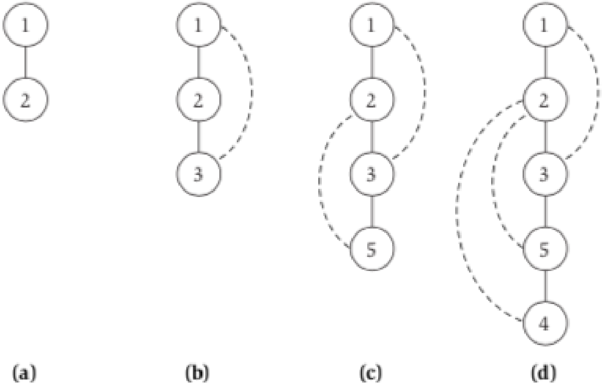


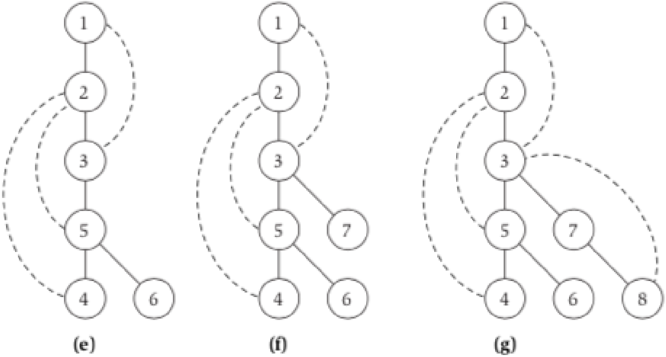
Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

DFS Tree - contd.

Now, consider the construction of the DFS tree, based on the DFS algorithm:



DFS Tree - contd.



DFS Tree Observations

Consider the following observation about an DFS tree relative to a BFS tree:

- A BFS tree root-to-leaf paths are short and narrow while those for a DFS tree are long and deep. This reflects the nature of the respective algorithms.

(3.6) - Relationship between the DFS algorithm and the structure of the DFS tree

(3.6) For a given recursive call DFS(u), all nodes that are marked "Explored" between the invocation and the end of this recursive call are descendants of u in T.

(3.7) - Structure of the DFS tree

(3.7) Let T be a DFS tree, let x and y be nodes in T, and let (x, y) be an edge of G that is *not* and edge of T. Then, one of x and y is an ancestor of the other.

(3.7) - Proof

Suppose that (x, y) is an edge of G that is not an edge of T and suppose without loss of generality that x is reached first by the DFS algorithm. When the edge (x, y) is examined during the execution of DFS(x), it is not added to T because y is marked "Explored". Since y was not marked "Explored" when DFS(x) was first invoked, it is a node that was discovered between the invocation and end of the recursive call DFS(x). It follows from (3.6) that y is a descendant of x.

The Set of All Connected Components

Until now, we've concerned ourselves with the connected component for a particular node, s. However, let's now consider the fact that all nodes in a graph, G, have a connected component.

What is the relationship between the connected components among the various nodes in a graph?

(3.8) - A highly structured relationship between connected components in a graph

(3.8) For any two nodes s and t in a graph, their connected components are either *identical* or *disjoint*.

(3.8) - Proof

Proof: We will complete this proof in two steps:

1. First, we'll show that if there exists a path between any two nodes s and t, then the connected components of s and t are *identical*.
2. Second, we'll show that if there does *not* exists a path between two nodes s and t, then their connected components are *disjoint*.

(3.8) - Proof (contd.)

1. Suppose there exists a path between any two nodes s and t. Now, let's suppose there exists a path between s and some other node, v, (and thus, v is a member of the connected component for s). Question: Is there a path from t to v? Answer: Yes, because we can make a path that consists first of the path from t to s, and then from s to v. So, this implies that v must also be in the connected component of t. So, for any node v in the connected component of s, v is also in the connected component of t, and vice versa by the same reasoning.

(3.8) - Proof (contd.)

1. Suppose there does *not* exist a path between two nodes s and t. Now, suppose, however, there exists a node v such that there exists a path from s to v and a path from t to v. Under this assumption of such a node, v, there *does* exist a path from s to t - namely, a path leading from s to v and then from v to t. Since our original assumption is that there is *no* path between s and t, such a node, v, cannot exist. Thus, if there is no path between two nodes, then they share no nodes in common in their respective connected components. Thus, the connected components for s and t are disjoint.