Basic Graph Theory (BFS, DFS, and Floodfill)

Albert Gural

October 7, 2011

1 Introduction

USACO Bronze contests will almost always have at least one or two problems relating to Breadth First Search (BFS), Depth First Search (DFS), or Floodfill. The concepts are relatively simple, and their uses are numerous as long as you know when to implement them. In order to fully understand BFS, DFS and floodfill, we need to look at the basic concepts first.

2 Graphs

A **graph** is a network containing **vertices** (aka nodes) and **edges**. Each vertex is connected to other vertices through edges. Edges can be directed (meaning there's a path from vertex A to vertex B, but not vice versa), or undirected. Edges can also have weights, meaning there's some cost associated with traveling from vertex A to vertex B. For bronze level competitions, these properties of graphs are generally not needed.

In many cases, you'll be working with **trees**, a special type of connected graph with N nodes and $N-1$ edges. Trees contain no cycles, meaning you can't get from one node back to itself by going through a different set of edges. This property will be very useful for some of our algorithms. Note: The topmost node is called the "root." Each node that points to lower nodes is a "parent" of the lower nodes - "children." Nodes with no children are known as "leaves."

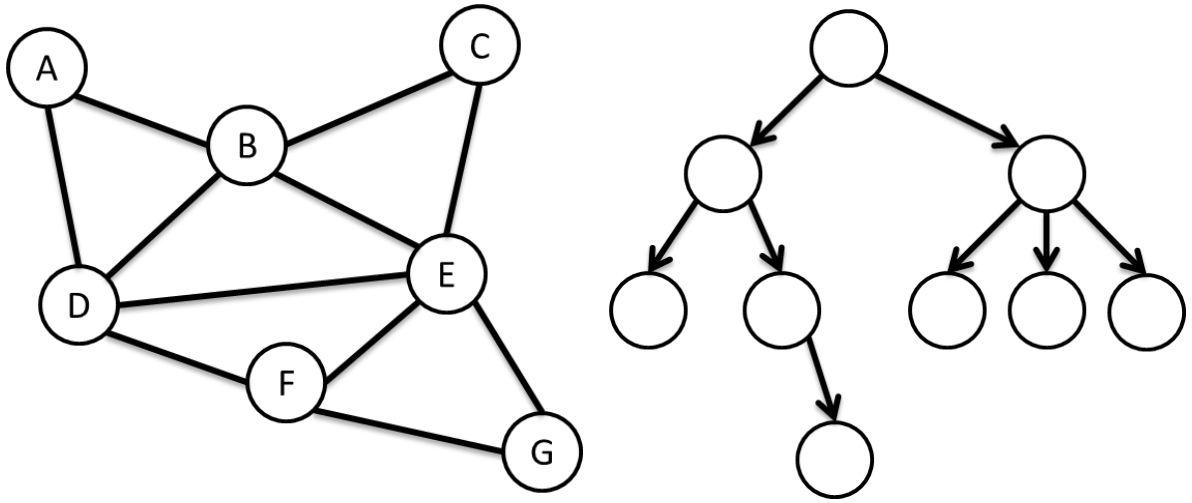

Figure 1: An Unweighted Undirected Graph (left) and a Tree Structure (right)

It often becomes a problem that you'll want to search for a solution or element in a graph. A major problem is how to traverse the nodes, since each node only knows information about its neighbors. There are several methods of traversal you can use to find a solution in a graph. The best method is highly dependent on the problem statement.

3 Depth First Search

Depth First Search or DFS is a method of traversing every element in a tree (or graph, if you keep track of traversed nodes) by recursively visiting the first child, then second, etc, from the root node.

The idea is we'll start at the root, then keep track of both children and go to the first child, treating it as a new root, and repeating the process until we reach a leaf. Then, we'll go back one step and go to the second child of the parent of the leaf and continue traversal. This process is recursive and continues until all nodes have been traversed.

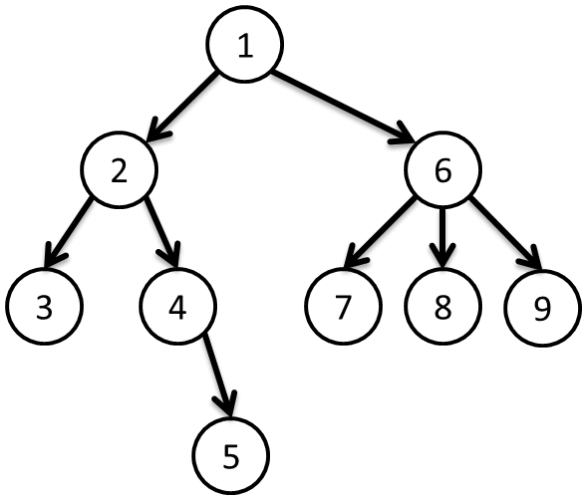

Figure 2: Node Traversal With Depth First Search

3.1 Recursion

The pseudocode is fairly simple. Suppose we simply want to traverse the nodes. Each node knows its children (in a list `node.children`). The following pseudocode demonstrates the simplicity of DFS. Try tracing the path this program will traverse the nodes of a tree:

```
call DFS(root)
procedure DFS(node):
    for child in node.children:
        call DFS(child)
    return
```

Can you modify this program to allow it to traverse a graph that contains cycles?

3.2 Stacks

Computers implement recursion through stacks (Last In First Out [LIFO] structure), so it's no surprise that stacks can be used to implement DFS. Although in some cases using a stack data structure might be faster than recursion, it's usually not worth the (minimal) additional coding time. See if you understand the pseudocode:

```
stack.push(root)
while stack not empty:
    node := stack.pop()
    for child in node.children:
        stack.push(child)
```

4 Breadth First Search

If you noticed, DFS traverses down the tree, then across. Alternately, you can traverse the tree by layer using Breadth First Search (BFS).

To do this, create a list of nodes to traverse and for each node in the list, add its children onto the end of the list. It should be clear why this will traverse the tree by layers.

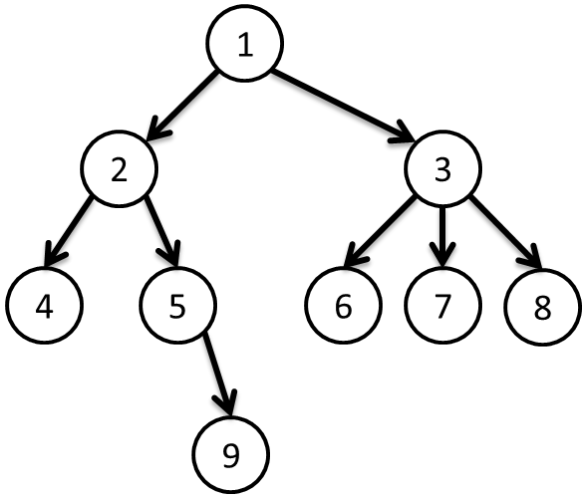

Figure 3: Node Traversal With Breadth First Search

4.1 Queues

Much like stacks are used for DFS, queues (First In First Out [FIFO] structure) are used for Breadth First Searching. Essentially, queues act exactly like the list described above: the first element that goes in is the first element traversed to in order to find its children. Pseudocode for traversing in a BFS manner might look like this:

```
queue.push(root)
while queue not empty:
    node := queue.pop()
    for child in node.children:
        queue.push(child)
```

Looks fairly familiar doesn't it? Again, this method only works for graphs without cycles. Can you modify it to work for any undirectional connected graph? This is very useful in many USACO problems.

5 DFS or BFS?

It's very important to know when to use BFS or DFS. While both methods will eventually find a solution, chosing the right method can be the difference between timing out and getting a program that executes in just a few milliseconds.

Suppose you have a graph and you want to find the closest element to some start node that solves a problem. Should you use BFS or DFS?

If you have a binary tree and want to find if any element equals 2, should you use BFS or DFS?

6 Depth First Search with Iterative Deepening

In some cases, you'll want a combination of DFS and BFS. This is called Depth First Search with Iterative Deepening. It works in the same way as DFS, but instead of recurring until you hit a leaf, stop after a certain recursion level k. Start with $k = 1$. If no solution is found, iterate $k = k + 1$, and search for a solution again. Continue until a solution is found.

Initially, this method seems to take longer than BFS. Why do you think this method might be more beneficial than BFS in certain cases? Can you think of problems in which using DFSID is better than using BFS?

7 Floodfill

Floodfill is another common bronze-level problem solving algorithm. Floodfill has to do with finding a traversal method to fill in connected parts of a graph. It's essentially a modified BFS where at each traversed node, you mark - "fill" that node (usually with the traversal distance from the start node). Floodfill can be used to find the shortest path from one point to another in an undirected unweighted graph.

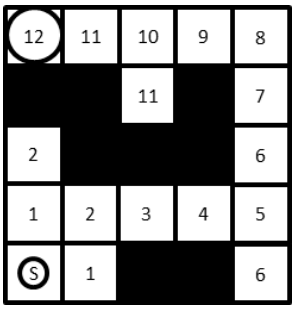

Figure 4: Floodfill Used to Solve a Maze Problem

8 Problems

1. (Rob Kolstad, 2009) Given an $X \times Y$ board containing obstacles and a knight that can only move like the chess piece, find the minimum number of moves necessary to reach a certain destination on that board.

2. (Traditional) N pastures are connected by paths in a tree structure. You know what pastures a given pasture connects to, and you have two target pastures A and B. What is the minimum number of paths that need to be traveled in order to get from A to B?

3. (Jeffrey Wang, 2007) Farmer John's pastures, arranged in an $X \times Y$ grid, have been partially flooded, forming lakes. Given that you know which pastures are flooded, what is the area of the largest lake? (The size of a lake is determined by the number of horizontally or vertically adjacent flooded pastures it contains.)

4. (Neal Wu, 2008) Bessie is caught at the lower left corner of an $X \times Y$ grid forest with various obstacles in her way. Given that she wants to reach the upper right corner, and that she never goes back to a point she's already visited, how many paths can she take?

5. (Sweet Tea Dorminy, 2010) An $X \times Y$ board resembling a maze (with paths and walls) contains paired teleports which work in the following way: If you step on a teleport marked by some letter, you will be instantly teleported to the corresponding letter teleport elsewhere on the board. Given a start location, what is the minimum number of steps needed to leave the maze?