Graph Traversals: Breadth-First Search and Depth-First Search

CMPSC 465 – CLRS Sections 22.2 and 22.3

I. The Idea

Recall how in rooted trees had traversals – inorder, preorder, and postorder – to visit all of the vertices in some particular fashion. We have analogs in graphs. In our study, we’ll always use a source vertex and traverse the graph from there. Two kinds of traversals are

- **breadth-first search** (BFS), which visits nodes in layers of those one step from the source first, then two steps from the source, etc.
- **depth-first search** (DFS), which visits nodes by going as deep into the graph from the source first, then backtracking and continually probing as deep as possible each time

Let’s look at a high-level view of both algorithms using the Mini-PSU Graph below:

![](image)

II. Colors and Attributes of Vertices for BFS

First, let’s say we’ll use s at the source vertex for a breadth-first search.

As would be implied by being a traversal algorithm, BFS visits many vertices of a graph. We visit each vertex more than once, but different visitations mean different things. Let’s begin by classifying vertices in one of two ways:

- **undiscovered** – those vertices in the graph we haven’t visited at all in the course of the algorithm so far
- **discovered** – those vertices we have visited so far

Easy enough. But it helps to make a further distinction within the set of discovered vertices. Let’s look at an illustration:

![](image)

We’ll, thus, make another distinction, and give each vertex a color to clarify the three different classes:

- **WHITE** vertices are those that are undiscovered
- **GRAY** vertices are those that are discovered, but we have yet to fully examine their adjacency lists
- **BLACK** vertices are those that are discovered and s.t. that we have also discovered all vertices in their adjacency lists
As we go, it will also be helpful to maintain a few other attributes of each vertex. (As far as I’m concerned, the book uses stupid names for some of these, so we’re going to do this in a more descriptive, yet still concise, way.) The three attributes each vertex $u$ will have during a BFS are:

- **color** – as defined above
- **dist** – the distance from the source vertex $s$ to $u$ (The book calls this $d$. Fair enough, but lazy.)
- **pred** – the predecessor to $u$ in the search (The book calls this $\pi$. I’m not here to confuse you.)

In the implementation, we’ll use a queue to maintain all of the **GRAY** vertices. Some notes:

- Recall: How do we traverse a queue?
- Why is this useful for BFS and **GRAY** vertices?

## III. The BFS Algorithm

Here is the pseudocode for BFS from a source vertex $s$ in a graph $G$, where adjacency lists (denoted $G.Adj[u]$ for a vertex $u$) are used to store adjacencies:

**BR**EADTH-FIRST-SEARCH($G, s$)

\[
\begin{align*}
&\text{for each vertex } u \in V(G) - \{s\} \quad \text{// initialize tracking attributes for all vertices but source } s \\
&\quad u.color = \text{WHITE} \\
&\quad u.dist = \infty \\
&\quad u.pred = \text{NIL} \\
\end{align*}
\]

$s.color = \text{GRAY}$ // initialize tracking attributes for source $s$

$s.dist = 0$

$s.pred = \text{NIL}$

$Q = \emptyset$ // set up queue of gray vertices, those on the frontier we’ve seen but not traversed all adjacent vertices; $s$ fits this to start

\[\text{ENQUEUE}(Q, s)\] // explore all the adjacencies to all vertices in FIFO ordering

\[
\text{while } Q \neq \emptyset \\
\text{for each } \ v \in G.Adj[u] \text{ s.t. } v.color = \text{WHITE} \\
\quad v.pred = u \\
\quad v.dist = u.dist + 1 \\
\quad v.color = \text{GRAY} \\
\quad \text{ENQUEUE}(Q, v) \\
\]

$u.color = \text{BLACK}$ // we’ve exhausted all neighbors to $u$ we could discover!
IV. BFS Example

Let’s trace the action of BFS on this graph:
V. BFS Analysis

**Question:** Let’s go back and look at a few snapshots of the queue during the BFS execution. What can we observe about the associated \( \text{dist} \) values?

Now, let’s look at the running time for an input graph \( G \):

- How long does it take to enqueue or dequeue a vertex? __________
- Consider an arbitrary vertex \( x \). How many times is it enqueued? Dequeued?

- How much total time do we invest in initializing vertices (coloring, predecessors, etc.)?
- How much total time do we spend enqueueing and dequeueing?
- The other contribution involves scanning adjacency lists. How much work is involved?

So, our total running time for BFS is ______________.

VI. BFS Spanning Trees

**Recall from 360:** What is a spanning tree for a graph \( G \)?

**Second recall question:** What is the precondition that guarantees that a graph \( G \) has a spanning tree?

So, notice that as breadth-first search goes, it builds a tree. Provided the right kind of input, this qualifies as a spanning tree, and we call it a **breadth-first spanning tree**. Using our example to the right (same as last example), what does the breadth-first spanning tree look like?

**Homework:** Read correctness of BFS on pp. 599-600; CLRS Exercises 22.2-1 and 22.2-2
VII. Attributes for DFS

As with BFS, we will attach various attributes to vertices as we traverse a graph with depth-first search. We’ll classify nodes as discovered or undiscovered and use the same WHITE/GRAY/BLACK coloring scheme. (Note that it’s possible to implement forms of BFS and DFS without the colors, but you still must mark nodes in some way. The colors make reasoning about the algorithms nice.) We also keep track of the predecessors of vertices we discover.

We keep track of the time when nodes are discovered (colored GRAY) and finished (colored BLACK). To do so, we maintain a global variable \textit{time}, which is initialized to 0 upon kicking off the DFS. Each vertex gets a time stamp when it is discovered and when it is finished, so timestamp values are integers between 1 and ________.

In total, there are five attributes each vertex \( u \) will have during a DFS:

- \textbf{color} – as defined above
- \textbf{pred} – the predecessor to \( u \) in the search
- \textbf{t\textsubscript{d}} – discovery time
- \textbf{t\textsubscript{f}} – finishing time

VIII. DFS Algorithm

We implement DFS via two methods:

- DFS – which takes in a graph \( G \) and a source vertex \( s \) and kicks off the DFS of \( G \) from \( s \).
- DFS-VISIT – which takes in a graph \( G \) and a WHITE vertex \( u \) and recursively processes the complete visit of \( u \) and any vertices “deeper” than it.

Here is the pseudocode for each:

DFS-VISIT(\( G, u \))

\{
    time = time + 1 \quad \text{ // white vertex } u \text{ has now been discovered}
    u.t\textsubscript{d} = time
    u.color = GRAY

    \textbf{for each} vertex \( v \in V(G).\text{Adj}[u] \) s.t. \( v.color = \text{WHITE} \)
    \{
        v.pred = u
        DFS-VISIT(\( G, v \)) \quad \text{ // recursively go deeper}
    \}

    u.color = BLACK \quad \text{ // } u \text{ is finished after all recursion completes}
    time = time + 1
    u.t\textsubscript{f} = time
\}

\textbf{DEPTH-FIRST-SEARCH(} \( G, s \))

\{
    \textbf{for each} vertex \( u \in V(G) - \{s\} \) \quad \text{ // initialize tracking attributes for all vertices but source}
    \{
        u.color = \text{WHITE}
        u.pred = \text{NIL}
    \}

    time = 0
    DFS-VISIT(\( G, s \))
\}
In summary,

- We explore edges out of the most-recently discovered vertex $v$ that still has unexplored edges leaving it
- Once all edges have been explored, we backtrack to explore edges leaving the vertex from which $v$ was discovered
- We continue this process until we have discovered all vertices reachable from the original source vertex.

**IX. DFS Example**

Let’s trace the action of DFS on this graph:
X. DFS Analysis

The analysis of the running time of DFS is very similar to that of BFS. Let’s break it down:

- Consider the `DEPTH-FIRST-SEARCH(G, s)` method. What is its running time, not counting recursive calls?

- Now consider `DFS-VISIT(G, u)`.
  - How many recursive calls happen in a call to `DFS-VISIT(G, u)`?

  - How much does each recursive call cost?

  - So how much does the call to `DFS-VISIT(G, s)` in `DEPTH-FIRST-SEARCH(G, s)` cost?

- Our total running time is thus…

XI. DFS Trees and Graph Structure Observations

As with BFS, if we run DFS on a connected graph, we can get out a depth-first spanning tree as a byproduct. In fact, we can classify every edge in a graph $G$ in one of four categories:

- Tree edges
- Back edges
- Forward edges
- Cross edges

In undirected graphs, the edges $(u, v)$ and $(v, u)$ are ambiguous. So, which classification do we choose when in doubt?

Let’s note a theorem:

**Theorem:** The depth-first tree for an undirected graph has only tree edges and back edges.
Example:
Run DFS on the following graph and classify each edge:

[Graph diagram]

Problem:

a. Pick any vertex \( v \) in the graph above that has descendants.
   i. What is \( v \)'s discovery time? _____ Finishing time? _____
   ii. List all descendants of \( v \) along with their finishing times.

b. Repeat for any other vertex.
   i. What is \( v \)'s discovery time? _____ Finishing time? _____
   ii. List all descendants of \( v \) along with their finishing times.

c. Observations?
Problem:
Back up to this graph and label each edge by its type:

Observations?

CLRS has something called the parenthesis theorem that relates to this; if you understand what happens above, that’s good enough!

Homework: CLRS Exercises 22.3-1, 22.3-2, 22.3-3, 22.3-7

Closing Note: You’ll notice in CLRS, BFS is presented as requiring a source and DFS is not. I chose to make the two consistent. Either algorithm could be extended to traversing the entire graph; one would simply need to go back and treat any unvisited vertex as a new source until we ran out of possible sources.