Readings

- [Lecture Notes Chapter 16: Graph Traversals: BFS & DFS]

Review: Graph Representations

Let $G = (V, E)$, where $|V| = n$ and $|E| = m$. In other words, $G$ contains $n$ vertices and $m$ edges.

Adjacency Matrix

We can represent $G$ with an $n \times n$ matrix $A$, where $A[i][j] = 1$ if there is an edge $(i, j)$ and 0 otherwise. The primary advantage of an adjacency matrix is that we can check whether or not there is an edge between two vertices in $O(1)$ time since all we have to do is index into the matrix. On the other hand, the matrix uses $\Theta(n^2)$ space, and it takes $\Theta(n)$ time to enumerate the neighbors of a vertex $v$ since we must iterate through an entire row in the matrix.

Adjacency List

We can also represent $G$ as an adjacency list, where we have an array $A$ of $|V|$ lists such $A[u]$ contains a LinkedList of vertices $v$ such that $(u, v) \in E$. (Note that if we have a directed graph, we can store two LinkedLists at index $u$, one for $u$’s in-neighbors and one for $u$’s out-neighbors.) The primary advantage of an adjacency list is that we only use $\Theta(n + m)$ space, which is better than the $\Theta(n^2)$ space usage of an adjacency matrix especially if $m \ll n^2$. Furthermore, it only takes $O(deg(v))$ time to enumerate the neighbors of a vertex $v$, but on the other hand, checking whether $(u, v) \in E$ also takes $O(deg(v))$ time since we have to traverse the LinkedList. In CIS 121, we usually use adjacency lists.

Review: Graph Traversals

We now look at two ways to traverse a graph.

BFS (Breadth First Search)

At a high-level, in BFS, we explore the graph in “layers,” so we explore “wide” before exploring “deep.” We begin at a node $v$ (layer 0), then explore the children of $v$ (layer 1), then the children of the nodes in layer 1 (which make up layer 2), etc. In other words, we explore all nodes at layer $L_i$ before exploring any nodes at layer $L_{i+1}$. Because we explore the graph in layers, we use a queue to implement this algorithm since it is FIFO, as shown in the pseudocode below. Note that the output of BFS is a BFS tree, and that as written, BFS is not guaranteed to visit every node in the graph.

The running time of BFS is $O(n + m)$. This is because we add and remove each node from the queue at most once, and for each node, we only examine its neighbors.
DFS (Depth First Search)

In DFS, we explore “deep” before exploring “wide.” We begin at some node \( v \) and examine its neighbors. When we encounter a neighbor that has not been visited yet, we visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack,” via returning from recursive calls, until we reach a node that still has unvisited neighbors. Because of this, DFS is recursive, but we can also implement it iteratively by using a stack since it is LIFO. The recursive pseudocode for DFS is below, and the iterative

Breadth-First Search

Input: A graph \( G = (V, E) \) implemented as an adjacency list and a source vertex \( s \).

Output: A BFS traversal of \( G \)

\[
\text{BFS}(G, s) \quad \text{for each} \ v \in V \ \text{do} \\
\quad \text{discovered}[v] = \text{FALSE} \\
\quad \text{parent}[v] = \text{NIL} \\
\]

Let \( Q \) be an empty Queue
\[
\text{Q.enqueue}(s) \\
\text{discovered}[s] = \text{TRUE} \\
\]

while \( Q \) is not empty do
\[
\quad v = \text{Q.dequeue()} \\
\quad \text{for each} \ u \in \text{Adj}[v] \ \text{do} \\
\qquad \text{if} \ \text{discovered}[u] = \text{FALSE} \ \text{then} \\
\qquad \quad \text{discovered}[u] = \text{TRUE} \\
\qquad \text{Q.enqueue}(u) \\
\qquad \text{parent}[u] = v \\
\]

DF (Depth First Search)

In DFS, we explore “deep” before exploring “wide.” We begin at some node \( v \) and examine its neighbors. When we encounter a neighbor that has not been visited yet, we visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack,” via returning from recursive calls, until we reach a node that still has unvisited neighbors. Because of this, DFS is recursive, but we can also implement it iteratively by using a stack since it is LIFO. The recursive pseudocode for DFS is below, and the iterative
pseudocode for DFS with a stack is the BFS pseudocode above, except instead of the queue we use a stack, instead of dequeue we use pop, and instead of enqueue we use push. From the pseudocode below, we know that DFS will visit every node in the graph, and so the output of DFS is a DFS forest. Furthermore, note that the pseudocode also introduces the concept of colors and finishing times, which makes DFS useful for other graph applications later on.

The running time analysis for DFS is similar to that of BFS; for each vertex \( v \), we iterate through its neighbors, ultimately yielding a runtime of \( O(n + m) \).

\[
\text{Depth-First Search}
\]

\textit{Input:} A graph \( G = (V, E) \) implemented as an adjacency list.

\textit{Output:} A DFS traversal of \( G \)

\[
\text{DFS}(G) \\
\text{for each } v \in V \text{ do} \\
\text{color}[v] = \text{WHITE} \\
\text{time} = 0 \\
\text{for each } v \in V \text{ do} \\
\text{if color}[v] = \text{WHITE then} \\
\text{DFS-VISIT}(G, v)
\]

\[
\text{DFS-VISIT}(G,u) \\
\text{time} = \text{time} + 1 \\
\text{d}[u] = \text{time} \\
\text{color}[u] = \text{GRAY} \\
\text{for each } v \in \text{Adj}[u] \text{ do} \\
\text{if color}[v] = \text{WHITE then} \\
\text{parent}[v] = u \\
\text{DFS-VISIT}(G,v) \quad \text{// go deeper into graph} \\
\text{color}[u] = \text{BLACK} \quad \text{// all u’s neighbors explored} \\
\text{time} = \text{time} + 1 \\
\text{f}[u] = \text{time}
\]

\textbf{Problems}

\textbf{Problem 1}

Design an \( O(n + m) \) algorithm to find the shortest path between nodes \( u \) and \( v \) in a connected, unweighted graph.

\textbf{Solution}

\textbf{Algorithm:} Run BFS starting from \( u \); for each node \( x \) that is visited, maintain a “parent” pointer to its parent node (the node we visit \( x \) from). Once we visit \( v \), stop and backtrack through these parent pointers until we reach \( u \), outputting this path as the shortest path. For example, by backtracking, we could
see that \( v \)'s parent was \( d \), \( d \)'s parent was \( c \), and \( c \)'s parent was \( u \), so the path outputted would be \( u - c - d - v \).

**Proof of Correctness:** The correctness of this algorithm follows from the lemma that BFS outputs the shortest paths in an unweighted graph. That is, for a vertex \( v \) in layer \( L_i \), \( i \) represents the length of the shortest path from the source \( u \) to \( v \). We prove this lemma via strong induction on \( i \):

**Base Case:** If \( i = 0 \), then by design, \( L_0 \) only contains the source vertex \( u \). The shortest path from \( u \) to \( u \) has length 0, so the claim holds.

**Induction Hypothesis:** Assume that for all vertices \( v \) such that \( v \) is in layer \( L_k \), where \( k < i \), that \( k \) is the length of the shortest path from \( u \) to \( v \).

**Induction Step:** We want to show that the claim holds for some arbitrary vertex \( v \) in layer \( L_{k+1} \). By design, \( v \) was discovered from another vertex \( w \), such that there is an edge between \( w \) and \( v \). Since \( w \) is in layer \( L_k \), by IH, we know that the shortest path from source \( u \) to vertex \( w \) has length \( k \), so by adding the edge from \( w \) to \( v \), we get a path of length \( k + 1 \) from source \( u \) to vertex \( v \). We want to show that this path is a shortest path from \( u \) to \( v \). Assume for the sake of contradiction that some other path \( u - \cdots - w' - v \) is the true shortest path, meaning that it must have length \( l \leq k \) and that \( w' \) is reachable from \( u \) by a shortest path with length less than or equal to \( k - 1 \). Thus, we know that vertex \( w' \) is in some layer \( L_{l-1} \), where \( l - 1 \leq k - 1 \). By design, this means that \( w' \) was removed from the BFS queue before \( w \), implying that when we processed \( w' \), we would have discovered \( v \). However, this means we would have added \( v \) to layer \( L_l \), and since \( l \leq k \), we have reached a contradiction because we stated that \( v \) was an arbitrary vertex in layer \( L_{k+1} \). Therefore, we have shown that the length of the shortest path from \( u \) to \( v \) is in fact \( k + 1 \), completing our Induction Step and thus our proof.

From the lemma, we know that BFS correctly computes the shortest paths from the source in an unweighted graph. In our algorithm, note that once we visit \( v \), we stop and backtrack through its parent pointers until we reach \( u \), so we just backtrack through the layers to yield the shortest path from \( u \) to \( v \). Hence, outputting this path correctly yields the shortest path from \( u \) to \( v \).

**Runtime Analysis:** Running BFS takes \( O(m + n) \) time, and to output the shortest path, we backtrack no more than \( O(n) \) times since the longest path in a graph has \( n - 1 \) edges. Therefore, our algorithm runs in \( O(m + n) \) time.

**Problem 2**

The CIS Department wants to assign prerequisites to their \( n \) classes. They have a list of \( m \) prerequisite pairings, such as CIS 110 being a prerequisite for CIS 120. Given the list of prerequisite pairings, design an algorithm that determines if this list of pairings is a valid list of pairings. For example, (CIS 110, CIS 120), (CIS 120, CIS 121), (CIS 121, CIS 110) is not a valid list of prerequisites. What is the runtime of your algorithm?

**Solution**

**Algorithm:** Construct a graph \( G \), where each of the vertices is one of the \( n \) classes in the CIS department and there is a directed edge \((u, v)\) if \( u \) is a prerequisite for \( v \). Run DFS on \( G \). If the DFS traversal finds a back edge during its execution, return that this list is invalid. Otherwise, return that this list is valid.

**Proof of Correctness:** If \( u \) is a prerequisite for \( v \), then by definition, we know that students must take course \( u \) before taking course \( v \). Directing an edge from \( u \) to \( v \) thus translates this exact information and nothing more, so \( G \) sufficiently translates all of the information given to us. Observe that a list of prerequisites is valid iff \( G \) does not contain a cycle. Now, we want to show that \( G \) has a cycle iff a DFS traversal finds a back edge:
(⇒) If $G$ has some cycle $C$, then let $v$ be the first vertex on this cycle discovered by DFS and let $(u, v)$ be the edge directly preceding $v$ on $C$. When $v$ is discovered, we know that all nodes in $C$ are still undiscovered and colored white. By design, $v$ is an ancestor of $u$ in the DFS forest, so $(u, v)$ is a back edge by definition.

(⇐) If there is a back edge $(u, v)$, then by design, we know that there is a path from $v$ to $u$ made of tree edges. Concatenating this path from $v$ to $u$ with the back edge $(u, v)$ creates a cycle.

Therefore, since $G$ has a cycle iff a DFS traversal finds a back edge, we know that our algorithm is correct.

**Runtime Analysis:** Constructing $G$ as an adjacency list takes $O(n + m)$ time. Running DFS on $G$ and determining if there are any back edges takes $O(n + m)$ time as well, so this algorithm runs in $O(n + m)$ time.