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 pseudocode is also provided.

**Breadth-First Search**

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

*Output:* A BFS traversal of \( G \)

\[
\text{BFS}(G, s)
\]

for each \( v \in V \) do
    discovered\([v]\) = FALSE
    parent\([v]\) = NIL

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

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

---

\[
\text{BFS}(G, s)
\]

for each \( v \in V \) do
    discovered\([v]\) = FALSE

discovered\([s]\) = TRUE
L\( [0] \) = \{s\}
i = 0

while L\([i]\) is not empty
    Let L\([i+1]\) be a new list
    for each \( v \in L[i] \) do
        for each \( u \in \text{Adj}[v] \) do
            if discovered\([u]\) = FALSE then
                discovered\([u]\) = TRUE
                parent\([u]\) = \( v \)
                L\([i+1]\).append\((u)\)
        
    i = i + 1

**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...
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 $O(n + m)$.

---

**Depth-First Search**

*Input*: A graph $G = (V, E)$ implemented as an adjacency list.

*Output*: A DFS traversal of $G$

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

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

---

**Problems**

**Problem 1**

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

**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
(CIS 120, CIS 121), (CIS 121, CIS 110) is not a valid list of prerequisites. What is the runtime of your algorithm?