CS 5084 - Introduction to Algorithms: Design and Analysis

Section 3.3 Implementing Graph Traversal Using Queues and Stacks

Joe Johnson

Table of Contents

Table of Contents

Parameters

Important parameters for the graph, $G = (V, E)$:

- $n = |V| \implies$ the number of nodes in G
- $m = |E| \implies$ the number of edges in G
- In many cases G is connected, in which case we know from (3.1) that the number of edges, m must be at least $n - 1$, i.e., $m \geq n - 1$.
    - Recall that (3.1) says that the number of edges in a tree, T, with n nodes is exactly $n - 1$ edges.

Calculating Time Complexity

Given m and n, how do we analyze the time complexities for the various algorithms?

- Is $O(m^2)$ better than $O(n^3)$?
- Since we won't know the relative magnitudes of m and n, our time complexities will be expressed in terms of both variables...

Methods for Representing a Graph

There are two basic ways to represent graphs:

1. adjacency matrix
2. adjacency list

Adjacency Matrix

Definition: Consider a graph $G = (V, E)$ with n nodes and assume the set of nodes is $V = \{1, \ldots, n\}$. The *adjacency matrix,* A for G is an $n \times n$ matrix where $A[u, v]$ is equal to 1 if the graph contains the edge (u, v) and 0 otherwise.

If the graph is undirected, the matrix is symmetric, meaning that $A[u, v] = A[v, u]$ for all nodes $u, v \in V$.

Adjacency Matrix

Advantage of an adjacency matrix:

- Checking whether edge, $(u, v) \in E$ can be done in constant time.

Disadvantages of an adjacency matrix:

- Space - The adjacency matrix requires $\Theta(n^2)$ space.
- The process of examining all edges incident with a particular node (frequently employed) requires $\Theta(n)$ time, since we must traverse all elements of the adjacency matrix row corresponding to that node. However, there may be *much fewer* nodes than n (i.e., the matrix is sparse). So, even though technically this process is $\Theta(n)$ using other approaches, from a practical perspective (i.e., in the *average* case) we should be able to do better than the worst case of n examinations for *every* node.

Adjacency List

Definition: Consider a graph $G = (V, E)$ with n nodes and assume the set of nodes is $V = \{1, \dots, n\}$. The adjacency list, *Adj* is an array consisting of n elements where each element, $Adj[u]$, corresponds to a node, u, in G and for which at element, u, there is a *list* of the nodes to which node v has edges.

In general, this works better for sparse graphs than an adjacency matrix.

Comparison of Adjacency List and Adjacency Matrix

Comparison of Adjacency List and Adjacency Matrix:

- Space:
    - Adjacency matrix: $O(n^2)$
    - Adjacency list: $O(m+n)$
        - Justification: There are n nodes, and thus, n elements in the array. For each edge, (u, v), we have two nodes added to the array - node v at element u in the array, and node u at element v in the array. Thus, for m edges, we have $2m$ elements in the array. Thus, our space requirement: $n + 2m = O(m + n)$.

Comparison of Adjacency List and Adjacency Matrix (contd.)

- Check whether an edge (u, v) exists in the graph:
  - Adjacency matrix: $O(1)$
  - Adjacency list: $O(n_v)$, where n_v is the *degree of* v, i.e., the number of edges incident with node v.
- Examine neighbors to a node:
  - Adjacency matrix: $O(n)$
  - Adjacency list: $O(n_v)$, i.e., constant time per neighbor (better!)

The Sum of degrees in a Graph

During the analysis of algorithms involving graphs, the *sum of the degrees in a graph*, or $\sum_{v \in V} n_v$, is a quantity that arises frequently. What is this sum?

(3.9) $\sum_{v \in V} n_v = 2m$

Proof: The degree of each node, v, is the number of *edges* incident with v. Each edge $e = (v, w)$ in the graph contributes exactly twice to the overall sum: once for node n_v and once for node n_w. Since the number of edges is m, we have that this sum is $2m$.

(3.10) Space Requirements for Adjacency Matrix and Adjacency List

(3.10) The adacency matrix representation of a graph requires $O(n^2)$ space, while the adjacency list requires only $O(m+n)$ space.

Note that since $m \leq n^2$, the bound $O(m+n)$ is never worse than $O(n^2)$ and much better, importantly, when the underlying graph is *sparse*, i.e, where $m << n$.

Table of Contents

Queues and Stacks

It is commonly the case that an algorithm needs to process a set of elements one at a time, thereby requiring a traversal through that set.

Two data structures that provide straight forward mechanisms for doing this are:

- Queue - FIFO.
- Stack - LIFO.

Both of these can easily be implemented using a doubly-linked list, which allows for addition/removal of elements in $O(1)$ time in both data structures.

Table of Contents

Implementing BFS - Important Items

A few important points about implementing BFS:

- The adjacency list data structure is ideal for implementing BFS.
- The algorithm examines the edges leaving a given node one by one.
- When we are scanning the edges leaving u and come to an edge (u, v), we need to know whether node v has been previously discovered by the search. $\implies$ we maintain an array, *Discovered* of length n and set $\mathit{Discovered}[v] = \mathit{true}$ as soon as the search first sees v.

Implementing BFS - Important Items - contd.

Important items for implementing BFS (contd.):

- The algorithm constructs layers of nodes, $L_1, L_2, L_3, \ldots$ where L_i is the set of nodes at distance i from the source, s.
- we have a list $L[i]$ for each $i = 0, 1, 2, \ldots$

Implementing BFS - Algorithm

```
BFS(s):
  Set Discovered[$s$] = true and Discovered[$v$] = false for all other $v$
  Initialize $L[0]$ to consist of the single element $s$
  Set the layer counter $i = 0$
  Set the current BFS tree $T = \emptyset$
  While $L[i]$ is not empty
    Initialize an empty list $L[i+1]$
    For each node $u \in L[i]$
      Consider each edge $(u, v)$ incident to $u$
      If Discovered[$v$] = false then
        Set Discovered[$v$] = true
        Add edge $(u, v)$ to the tree $T$
```

Implementing BFS - Algorithm Analysis

(3.11) This implementation of the BFS algorithm runs in $O(m+n)$, i.e., linear in the input size, if the graph is given by the adjacency list representation.

Implementing BFS - Algorithm Analysis (contd.)

Proof: Consider the time spent in the For loop for each node, u. Given we're using an adjacency matrix, the amount of time required to access each edge, (u, v) incident with u, is $O(1)$. Further, the amount of time necessary to check whether node v has been discovered, i.e. checking $Discovered[v]$ is also $O(1)$. Finally, the number of nodes which we must traverse for u is n_u. Thus the time spent in each iteration of the For loop for each node u is $O(n_u)$. Thus, the total amount of time spent in the For loop across *all* nodes is $O(\sum_{v \in V} v)$. But we know from (3.9) that $\sum_{v \in V} v = 2m$. Thus, we have that the time complexity for the for loop is $O(m)$. We also have to set up the lists and the array, *Discovered*, each of which require $O(n)$. Thus, total runtime $= O(m + n)$.

Implementation of BFS Using a Queue

An alternative implementation of BFS and actually, the implementation that is used in the “real world” is to use a queue instead of building the lists, $L_1, L_2, \ldots$. This approach inherently provides for exploring the nodes in the order in which they are discovered, as BFS requires.

Table of Contents

Implementing DFS - Important Items

- DFS maintains the nodes to be explored in a stack, as opposed to a queue (for BFS).
- While exploring a node, u, the neighbors of u are each pushed onto a stack for later processing.
- Placing neighbors on top of a stack (instead of at the back of a queue) has the effect of traveling down a single path as far from starting node s as quickly as possible, as opposed to exploring all nodes in layers of a fixed distance, i from s before moving on to those of distance $i + 1$ from s, and so on.

Implementing DFS - Important Items (contd.)

- An important distinction must be made between *visiting* a node vs. *exploring* a node.
  - discover a node, v - we find an edge leading to node v.
  - explore a node, v - we scan all edges incident with node v.

Implementing DFS - Algorithm

```
DFS(s):
  Initialize $S$ to be a stack with one element $s$
  While $S$ is not empty
    Take a node $u$ from $S$
    If Explored[$u$] = false then
      Set Explored[$u$] = true
      For each edge $(u, v)$ incident to $u$
        Add $v$ to the stack $S$
      Endfor
    Endif
  Endwhile
```

(3.12) DFS Implementation

(3.12) This algorithm implements DFS in the sense that it visits the nodes in exactly the same order as the recursive DFS procedure we gave in section 3.2, except that each adjacency list is processed in reverse order).

(3.13) DFS Implementation - Algorithm Analysis

(3.13) This implementation of the DFS algorithm runs in $O(m+n)$, i.e., linear time with respect to the size of the input, if the graph is given by the adjacency list representation.

A proof of this can be constructed using very similar reasoning to the proof provided for the DFS algorithm implementation, (3.11).

Table of Contents

Finding the Set of All Connected Components

An algorithm for finding the set of *all* connected components in a graph, G:

1. Start with an arbitrary node, s, and find its connected component using either BFS or DFS.
2. Find a node v where v does not exist in the connected component of s, if one exists, and find the connected component of v.
3. Repeat this process until there all nodes in G have been visited.