Lecture 26: Graphs: Traversal (Part 1)

10:00 AM, Apr 2, 2018

Contents

1 Introduction .............................. 1
  1.1 Definitions .............................. 2
  1.2 Representations .......................... 3

2 Graph Traversal ........................... 5
  2.1 BFS and DFS on Trees ................... 5

Objectives

By the end of this lecture, you will know:

• what a directed graph is;
• how to represent directed graphs in a program; and
• the advantages and disadvantages of different representations

By the end of this lecture, you will be able to:

• traverse the simplest kind of graph—trees—in breadth-first order

1 Introduction

A graph consists of a set of vertices, or nodes, and a set of edges, or links, among those nodes, often annotated with weights. As a modeling tool, graphs are pervasive, both inside and outside computer science. They are used to model telecommunication networks (e.g., the Internet), social networks (e.g., Facebook), maps (e.g., a road map), etc. Here are some more details about how graphs can be used to model these things, together with examples of the kind of problems we can solve using graphs as a modelling tool:

• Internet:
  – nodes: web pages
  – links: hyperlinks pointing from one web page to another
  – application: which are the most authoritative web pages?

• Facebook:
In the first part of this lecture, we present graphs mathematically and discuss two viable computational representations of graphs. In the second part, we introduce some basic graph algorithms. Overall, the next few lectures should give you an understanding of how to work with graphs and a flavor of the kind of things you will learn about should you choose to take CSCI 1570.

Earlier versions of these notes were based on the *Introduction to Algorithms* textbook, by Cormen, Leiserson, Rivest, and Stein (2001). The current version is only very loosely based on this source.

### 1.1 Definitions

A *graph* $G$ is a pair $(V, E)$, where $V$ is a finite set of vertices (sometimes called nodes) and $E$ is a finite set of edges (sometimes called links). By convention, $n = |V|$ and $m = |E|$.

In a *directed* graph, $E$ is a set of ordered pairs, while in an *undirected* graph, $E$ is a set of unordered pairs. For example, the pair of sets

$$\{\{1, 2, 3, 4, 5\}, \{(1, 2), (1, 5), (2, 1), (2, 4), (3, 4), (3, 5), (4, 5)\}\}$$

is a directed graph, while the pair of sets

$$\{\{1, 2, 3, 4, 5\}, \{\{1, 2\}, \{1, 5\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}\}$$

is an undirected graph.

An edge from a vertex to itself is called a *self-loop*. In an undirected graph, the maximum number of edges (not including self-loops) is $\binom{n}{2} = n(n-1)/2$, the number of different ways to choose 2 items among $n$. Including self-loops, it is $\binom{n}{2} + n = n(n + 1)/2$. In a directed graph, the maximum number of edges including self-loops is $n^2$. Not including self-loops, it is $n(n - 1)$. In all cases, $m \in O(n^2)$. 

---

1. nodes: people
2. links: friendships
3. application: how many degrees of separation are there between me and Barack Obama?

- Road Map:
  - nodes: intersections
  - links: streets and highways
  - weights: distances
  - application: what is the fastest way to drive from Providence to New York?
We say that an edge \((u, v)\) is *incident* upon the vertices \(u\) and \(v\). We also say that \(u\) is an (immediate) *predecessor* of \(v\) and that \(v\) is an (immediate) *successor* of \(u\). In addition, \(u\) and \(v\) are called *neighbors*.

In an undirected graph, the *degree* of a vertex is the number of edges incident upon it. In a directed graph, the *indegree* of a vertex is the number of edges entering it, while the *outdegree* of a vertex is the number of edges exiting it.

A *path* from vertex \(u_1\) to vertex \(u_2\) is a sequence of vertices \(\langle u_1 = v_0, v_1, \ldots, v_k = u_2 \rangle\) such that \((v_i, v_{i+1}) \in E\). By convention, the length of a path \(v_0, v_1, \ldots, v_k\) is \(k\); so the length of a path from a vertex to itself is 0. If \(v_0 = v_k\), then the path is a *cycle*.

If there is a path from \(u\) to \(v\), then the vertices \(u\) and \(v\) are said to be *connected*. If there is a path from \(u\) to \(v\) for all \(u, v \in V\), then the graph is said to be *connected*.

A graph with no cycles is called an *acyclic* graph. A *tree* is a connected, acyclic graph.

A directed graph with no cycles is called a *directed acyclic graph*, or a DAG, for short. DAGs can be used to represent jobs (vertices) that need to be scheduled when there are precedence constraints (edges) between (at least) some of them: e.g., you should tune your violin before practicing, but it doesn’t really matter if you do your CS 18 homework before or after practicing.

Often, the edges in a graph are annotated with *weights*; the weight on edge \((u, v)\) is denoted by \(w(u, v)\). Sometimes these weights are called *costs*, and denoted by \(c(u, v)\) instead.

### 1.2 Representations

There are two common ways to represent graphs. Perhaps the most straightforward is to use an \(n \times n\) adjacency matrix \(A\). In an unweighted graph, \(a_{ij} = 1\) if and only if \((i, j) \in E\); in a weighted graph, \(a_{ij} = w(i, j)\). In undirected graphs, \(A^T = A\): i.e., \(A\) is *symmetric* along the main diagonal. The size of an adjacency matrix is \(O(n^2)\).

Figure 3 shows an adjacency matrix representing the graphs in Figures 1 and 2. Observe that the matrix is indeed symmetric for the undirected graph. In some cases (particularly, where space is an issue), it may make sense to store an undirected graph as an array of arrays of increasing lengths rather than as a matrix, so as to avoid storing the same information twice.

For *dense* graphs, nearly all the cells in the adjacency matrix store data. But for *sparse* graphs, many of the cell’s values are 0. In this latter case, an adjacency set is more space efficient.

An *adjacency set* is an array of size \(n\), in which each cell \(i\) stores the set of vertices that \(i\) connects
Figure 3: (Left) An adjacency matrix representing the directed graphs in Figures 1 and 2. (Right) An adjacency matrix representing the undirected graphs in Figures 1 and 2.

to: that is, \( i \)'s set includes all vertices \( j \) such that \((i, j) \in E\). (In a weighted graph, the sets also store weights.) The size of an adjacency set is \( O(n + m) \).

A common implementation of an adjacency set is as an array of linked lists, as seen in Figure 5. Another popular choice is an array of hash tables. For weighted edges that are accessed in weight order, an array of heaps makes practical sense. Or, if the edges are ordered an array of binary search trees is a possibility.

![Nodes Linked Lists]

Figure 4: An adjacency set representing the undirected, unweighted graphs in Figures 1 and 2.

The choice of adjacency matrix vs. adjacency set is a prime example of the time vs. space trade-off which arises so often in computer science. In terms of time:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Adjacency Matrix</th>
<th>Adjacency Linked List</th>
<th>Adjacency Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there an edge from ( u ) to ( v )?</td>
<td>( O(1) )</td>
<td>( O(m) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>How fast is insertion and deletion of edges (checking for duplicates)?</td>
<td>( O(1) )</td>
<td>( O(m) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>What is the degree of vertex ( u )?</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

The time complexity for determining the degree of vertex \( u \) with an adjacency linked list assumes the data structure maintains a length attribute (rather than traverse the list to determine its length).

\(^1\)Comparable, in Java.
If we include the restriction that a vertex can have at most \( n \) edges, then the other two adjacency linked list time complexities would drop to \( O(\min\{m, n\}) \).

The time complexity for an adjacency hash table assumes a distribution of keys together with a hash function that uniformly distributes keys, and a hash table with sufficient capacity so that there are not too many collisions.

But what about space? The adjacency matrix always takes \( O(n^2) \) space: there’s an entry for each possible edge, i.e., each pair of nodes. This is true no matter how many edges the graph has. In contrast, the adjacency list may take up far less space if \( m \) is small compared to \( n \): there can only ever be \( m \) entries total across the lists for all nodes.

## 2 Graph Traversal

There are two principal ways to traverse a graph: breadth-first and depth-first. Suppose we are given a source (i.e., start) node and asked to traverse the entire graph (or at least, as much as can be reached!) from there. This is often cached in terms of searching for a particular sink (i.e., destination) node, but the general technique is far more broad.

The key idea underlying breadth-first traversal is to visit nodes in order of distance from the source, where distance is measured as the number of edges traversed (as opposed to edge costs, in a weighted graph). The key idea underlying depth-first traversal is to visit nodes as deeply as possible first, and then to backtrack until a node is discovered whose neighbors have not yet all been visited.

Breadth-first and depth-first traversals are often part of search routines (e.g., to search for a path from Providence to New York City in a graph that represents a road map), and we will refer to these algorithms as breadth-first search (BFS) and depth-first search (DFS), respectively.

### 2.1 BFS and DFS on Trees

To begin with, let’s look at an example of a tree and its corresponding breadth-first and depth-first traversals. While there are recursive formulations of both algorithms, here we will focus on their iterative variants.

![Figure 5: Example Tree](image)

BFS is implemented using a queue. First, enqueue the root node \( A \) onto the queue. Then, dequeue \( A \), and enqueue its children (typically, in left-to-right order) onto the queue. The queue now looks like \( B, C \). Next, dequeue \( B \), and enqueue its children onto the queue. The queue now looks like
C, D, E. Next, dequeue C, and enqueue its children onto the queue. The queue now looks like D, E, F, G. And so on. Indeed, BFS visits nodes in this tree in alphabetical order.

In contrast, DFS is implemented using a stack.

Here is how a DFS of the above tree proceeds. First, push the root node A onto the stack. Then, pop A, and push its children (typically, in left-to-right order) onto the stack. The stack now looks like B, C. Next, pop B, and push its children onto the stack. The stack now looks like D, E, C. Next, pop D, and push its children onto the stack. The stack now looks like H, I, E, C. Next, pop H; next, pop I. The stack now looks like E, C. Next, pop E, and push its children on to the stack. The stack now looks like J, K, C. And so on. Ultimately, DFS visits nodes in this tree (i.e., pops nodes off the stack) in the following order: A, B, D, H, I, E, J, K, C, F, L, M, G, N, O.

BFS and DFS for Trees

Here is pseudocode for these two types of search for a tree $G$ starting from node $v$. Notice that they are essentially the same algorithm, varying only in the choice of data structure used to store the “to do” list.

```java
DFT(G, v) {
    S = new Stack()
    S.push(v)
    while (!S.empty()) {
        v = S.pop()
        for (w in G.adjs(v)) {
            S.push(w)
        }
    }
}

BFT(G, v) {
    Q = new Queue()
    Q.add(v)
    while (!Q.empty()) {
        v = Q.getFirst()
        Q.removeFirst()
        for (w in G.adjs(v)) {
            Q.add(w)
        }
    }
}
```

Note that these versions are explicitly just for trees. We’ll talk about why that is in the next lecture.

---

2 There are actually multiple variations of DFS, depending on whether nodes are popped off the stack before or after its children are visited. The version we present here is called a “pre-order” traversal.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)