Introduction to Graphs
Graph Traversals

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, November 30, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
Some material contributed by Chris Hartman
Review
Unit Overview
Tables & Priority Queues

Major Topics
✓ § Introduction to Tables
✓ § Priority Queues
✓ § Binary Heap Algorithms
✓ § Heaps & Priority Queues in the C++ STL
✓ § 2-3 Trees
✓ § Other self-balancing search trees
✓ § Hash Tables
✓ § Prefix Trees
✓ § Tables in the C++ STL & Elsewhere

Idea #1: Restricted Table
Lots of lousy implementations

Idea #2: Keep a tree balanced

Idea #3: Magic functions
A special-purpose implementation: “the Radix Sort of Table implementations”

DONE
The Rest of the Course
Overview

Two Final Topics
✓  ▪ External Data
  ▪ Previously, we dealt only with data stored in memory.
  ▪ Suppose, instead, that we wish to deal with data stored on an external device, accessed via a relatively slow connection and available in sizable chunks (data on a disk, for example).
  ▪ How does this affect the design of algorithms and data structures?

▪ Graph Algorithms
  ▪ A graph models relationships between pairs of objects.
  ▪ This is a very general notion. Algorithms for graphs often have very general applicability.

This usage of “graph” has nothing to do with the graph of a function. It is a different definition of the word.
We consider data that are accessed via a slow channel. Typically, the channel transmits data in chunks: blocks. Thus, minimize the number of block accesses.

External Sorting: Merge Sort variant
- Stable Merge works well with block-access data.
- Use temporary files for the necessary buffers.

External Table Implementation #1: Hash Table
- Open hashing, with each bucket stored in a small fixed number of blocks where possible, works well.
External Table Implementation #2: B-Tree

A **B-Tree of degree** $m$ ($m \geq 3$) is a ceiling($m/2$) ... $m$ Tree.

- A node has ceiling($m/2$)–1 ... $m$–1 items.
- Except: The root can have 1 ... $m$–1 items.
- All leaves are at the same level.
- Non-leaves have 1 more child than # of items.
- The order property holds, as for 2-3 Trees and 2-3-4 Trees.
- Degree = max # of children = # of items in an **over-full** node.

2-3 Tree = B-Tree of degree 3. 2-3-4 Tree = B-Tree of degree 4.

Shown is a B-Tree of degree 7.

In practice, the degree may be much higher (for example, 50).
B-Tree algorithms are similar to those for a 2-3 Tree.

Example. Insert 40 into this B-Tree of degree 7.

An over-full node would contain 7 items.

Over-full nodes are blue.
There are a number of B-Tree variations. A common one: \textbf{B+ Tree}. This is the same as a B-Tree, except:

- Keys in non-leaf nodes are duplicated in the leaves, while maintaining the order property.
- Associated values are stored only in the leaves.
- Leaves are joined into an auxiliary Linked List. This minimizes the number of block accesses required for a traversal.

Each \textit{key} in a non-leaf is duplicated in a leaf. The order property holds.

The \textit{associated value} for key 84 is only stored here.
To the right is a Table dataset. Below on the left is a B-Tree holding this dataset. Below on the right is the corresponding B+ Tree. Both keys and associated values are shown.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>7</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
</tr>
</tbody>
</table>

Modern filesystems typically involve a B-Tree or variant internally.
- B+ Trees are a particularly common variant.

These trees are also used in relational-database implementation.
Introduction to Graphs
Introduction to Graphs
Definition

A **graph** consists of **vertices** and **edges**.
- An edge joins two vertices: its **endpoints**.
- 1 vertex, 2 vertices (Latin plural).
- Two vertices joined by an edge are **adjacent**; each is a **neighbor** of the other.

In a **weighted graph**, each edge has a **weight** (or **cost**).
- The weight is the resource expenditure required to use that edge.
- We typically choose edges to minimize the total weight of some kind of collection.

If we give each edge a direction, then we have a **directed graph**, or **digraph**.
- Directed edges are called **arcs**.
Introduction to Graphs
Applications

We use graphs to model:

- Networks
  - Vertices are nodes in network; edges are connections.
  - Examples
    - Communication
    - Transportation
    - Electrical
    - Worldwide Web (edges are links)

- State Spaces
  - Vertices are states; edges are transitions between states.

- Generally, situations in which objects are related in pairs:
  - Vertices are data-structure nodes; directed edges indicate pointers.
  - Vertices are people, edges indicate relationships (friendship?).
  - Vertices are tasks or events; edges join pairs that cannot occur at the same time (e.g., because of conflicting resource needs).
How do we represent a graph in a computer program?

**Adjacency matrix.** 2-D array of 0/1 values.
- “Are vertices $i$, $j$ adjacent?” in $\Theta(1)$ time.
- Finding all neighbors of a vertex is slow for large, sparse graphs.
  - *Sparse* graph: one with relatively few edges.

**Adjacency lists.** List of lists (arrays?). List $i$ holds neighbors of vertex $i$.
- “Are vertices $i$, $j$ adjacent?” in $\Theta(\log N)$ time if lists are sorted arrays; $\Theta(N)$ if not.
- Finding all neighbors can be faster.

Both adjacency matrices and adjacency lists can be generalized to handle digraphs.

---

$N$: the number of vertices (more on this soon)

Adjacency Matrix:

```
0 0 1 0 1
1 1 0 0 0
2 0 0 0 0
3 1 0 0 0
```
Introduction to Graphs
Representations [2/3] (Try It!)

For the following graph, write (a) the adjacency matrix, and (b) adjacency lists.

Answers on the next slide.
For the following graph, write (a) the adjacency matrix, and (b) adjacency lists.

Answers

(a)  
\[
\begin{bmatrix}
0 & 1 & 2 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
2 & 0 & 1 \\
\end{bmatrix}
\]

(b)  
\[
\begin{aligned}
0: & 1 \\
1: & 0, 2 \\
2: & 1 \\
\end{aligned}
\]

Green numbers are optional in the answer to part (a).
Introduction to Graphs
Analyzing Efficiency

When an algorithm takes a graph, what is our “n”? The number of vertices? The number of edges? Some combination?

We consider both the number of vertices and the number of edges.
- $N = \text{number of vertices}$
- $M = \text{number of edges}$

Adjacency matrices & adjacency lists are considered separately.

The total size of the input is:
- For an adjacency matrix: $N^2$. So $\Theta(N^2)$.
- For adjacency lists: $N + 2M$. So $\Theta(N + M)$.

Some particular algorithm might have order (say) $\Theta(N + M \log N)$. I use upper case ($N, M$) to make it clear that we are talking about vertices and edges, not the size of the input as a whole.

The “2” is because each edge corresponds to two entries in the adjacency lists—one for each endpoint of the edge.
Graph Traversals
Graph Traversals

Introduction

We covered Binary Tree traversals: preorder, inorder, postorder. We traverse graphs as well.

- Here, to **traverse** means to visit each vertex (once).
- Traditionally, graph traversal is viewed in terms of a “search”: visit each vertex searching for something.

Two important graph traversals.

- **Depth-first search (DFS)**
  - Similar to a preorder Binary Tree traversal.
  - When we visit a vertex, give priority to visiting its unvisited neighbors (and when we visit one of them, we give priority to visiting its unvisited neighbors, etc.).
  - Result: we may proceed far from a vertex before visiting all its neighbors.

- **Breadth-first search (BFS)**
  - Visit all of a vertex’s unvisited neighbors before visiting their neighbors.
  - Result: Vertices are visited in order of distance from start.
Graph Traversals
DFS [1/2]

DFS has a natural recursive formulation:

- Given a start vertex, visit it, and mark it as visited.
- For each of the start vertex’s neighbors:
  - If this neighbor is unvisited, then do a DFS with this neighbor as the start vertex.

Recursion can be eliminated with a Stack. And we can be more intelligent than the brute-force method.
Graph Traversals
DFS [2/2]

Algorithm DFS
- Mark all vertices as unvisited.
- For each vertex:
  - Do algorithm DFS’ with this vertex as start.

Algorithm DFS’
- Set Stack to empty.
- Push start vertex on Stack.
- Repeat while Stack is non-empty:
  - Pop top of Stack.
  - If this vertex is not visited, then:
    - Visit it.
    - Push its not-visited neighbors on the Stack.

TO DO
- Write a non-recursive function to do a DFS on a graph, given adjacency lists.

Done. See graph_traverse.cpp.

This part is all we need, if the graph is connected (all one piece). The above is only required for disconnected graphs.
Graph Traversals
BFS [1/2]

BFS is good for finding the shortest paths to other vertices.
Graph Traversals
BFS [2/2]

TO DO

- Modify our DFS function to do BFS.
  - BFS reverses the priority of neighbors vs. neighbors of neighbors.
  - Thus: replace the Stack with a Queue.

Done. See graph_traverse.cpp.
Graph Traversals
TO BE CONTINUED ...

*Graph Traversals* will be continued next time.