CSE 332 Autumn 2023
Lecture 18: Graphs

Nathan Brunelle

http://www.cs.uw.edu/332
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

<table>
<thead>
<tr>
<th>103</th>
<th>801</th>
<th>401</th>
<th>323</th>
<th>255</th>
<th>823</th>
<th>999</th>
<th>101</th>
<th>113</th>
<th>901</th>
<th>555</th>
<th>512</th>
<th>245</th>
<th>800</th>
<th>018</th>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Place each element into a “bucket” according to its 1’s place
RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases

- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>801</td>
<td>401</td>
<td>101</td>
<td>901</td>
<td>121</td>
<td>512</td>
<td>103</td>
<td>323</td>
<td>823</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>555</td>
<td>245</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>018</td>
<td>999</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Place each element into a “bucket” according to its 10’s place
RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases

- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

Place each element into a “bucket” according to its 100’s place
RadixSort

- **Radix**: The base of a number system
  - We’ll use base 10, most implementations will use larger bases

- **Idea**:
  - BucketSort by each digit, one at a time, from least significant to most significant

<table>
<thead>
<tr>
<th>018</th>
<th>101</th>
<th>103</th>
<th>113</th>
<th>121</th>
<th>245</th>
<th>255</th>
<th>323</th>
<th>401</th>
<th>512</th>
<th>555</th>
<th>800</th>
<th>801</th>
<th>823</th>
<th>901</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>101</td>
<td>103</td>
<td>113</td>
<td>121</td>
<td>245</td>
<td>255</td>
<td>323</td>
<td>401</td>
<td>512</td>
<td>555</td>
<td>800</td>
<td>801</td>
<td>823</td>
<td>901</td>
<td>999</td>
</tr>
</tbody>
</table>

Convert back into an array
RadixSort Running Time

• Suppose largest value is $m$
• Choose a radix (base of representation) $b$
• BucketSort all $n$ things using $b$ buckets
  • $\Theta(n + b)$
• Repeat once per each digit
  • $\log_b m$ iterations
• Overall:
  • $\Theta(n \log_b m + b \log_b m)$
• In practice, you can select the value of $b$ to optimize running time
• When is this better than mergesort?
Undirected Graphs

Definition: $G = (V, E)$

Vertices/Nodes

$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Edges

$E = \{(1,2), (2,3), (1,3), \ldots\}$
Definition: \( G = (V, E) \)

- Vertices/Nodes
  \[ V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \]
- Edges
  \[ E = \{(1,2), (2,3), (1,3), \ldots\} \]
Self-Edges and Duplicate Edges

Some graphs may have duplicate edges (e.g. here we have the edge (1,2) twice). Some may also have self-edges (e.g. here there is an edge from 1 to 1). Graph with Neither self-edges nor duplicate edges are called simple graphs.
Weighted Graphs

Definition: \( G = (V, E) \)

\[ w(e) = \text{weight of edge } e \]

Vertices/Nodes
\[ V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

Edges
\[ E = \{(1,2), (2,3), (1,3), \ldots\} \]
Graph Applications

• For each application below, consider:
  • What are the nodes, what are the edges?
  • Is the graph directed?
  • Is the graph simple?
  • Is the graph weighted?

• Facebook friends
• Twitter followers
• Java inheritance
• Airline Routes
Some Graph Terms

- **Adjacent/Neighbors**
  - Nodes are adjacent/neighbors if they share an edge
- **Degree**
  - Number of “neighbors” of a vertex
- **Indegree**
  - Number of incoming neighbors
- **Outdegree**
  - Number of outgoing neighbors
Graph Operations

• To represent a Graph (i.e. build a data structure) we need:
  • Add Edge
  • Remove Edge
  • Check if Edge Exists
  • Get Neighbors (incoming)
  • Get Neighbors (outgoing)
Adjacency List

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(n)$
Get Neighbors (incoming): $\Theta(n + m)$
Get Neighbors (outgoing): $\Theta(\text{deg}(v))$

$|V| = n$
$|E| = m$
Adjacency List (Weighted)

Time/Space Tradeoffs
Space to represent: $\Theta(n + m)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(n)$
Get Neighbors (incoming): $\Theta(?)$
Get Neighbors (outgoing): $\Theta(?)$

$|V| = n$
$|E| = m$
Adjacency Matrix

Time/Space Tradeoffs
Space to represent: $\Theta(\ ?)$
Add Edge: $\Theta(\ ?)$
Remove Edge: $\Theta(\ ?)$
Check if Edge Exists: $\Theta(\ ?)$
Get Neighbors (incoming): $\Theta(\ ?)$
Get Neighbors (outgoing): $\Theta(\ ?)$

$|V| = n$
$|E| = m$
Adjacency Matrix (weighted)

Time/Space Tradeoffs
Space to represent: $\Theta(n^2)$
Add Edge: $\Theta(1)$
Remove Edge: $\Theta(1)$
Check if Edge Exists: $\Theta(1)$
Get Neighbors (incoming): $\Theta(n)$
Get Neighbors (outgoing): $\Theta(n)$

$\mid V \mid = n$
$\mid E \mid = m$
Aside

• Almost always, adjacency lists are the better choice
• Most graphs are missing most of their edges, so the adjacency list is much more space efficient and the slower operations aren’t that bad
Definition: Path

A sequence of nodes \((v_1, v_2, \ldots, v_k)\)
s.t. \(\forall 1 \leq i \leq k - 1, (v_i, v_{i+1}) \in E\)

Simple Path:
A path in which each node appears at most once

Cycle:
A path which starts and ends in the same place
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$
Definition: (Strongly) Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$

Connected

Not (strongly) Connected
Definition: Weakly Connected Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from $v_1$ to $v_2$ ignoring direction of edges.

Weakly Connected

Weakly Connected
Definition: Complete Graph

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from $v_1$ to $v_2$
Graph Density, Data Structures, Efficiency

• The maximum number of edges in a graph is $\Theta(|V|^2)$:
  • Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  • Directed and simple: $|V|(|V|-1)$
  • Direct and non-simple (but no duplicates): $|V|^2$
• If the graph is connected, the minimum number of edges is $|V| - 1$
• If $|E| \in \Theta(|V|^2)$ we say the graph is dense
• If $|E| \in \Theta(|V|)$ we say the graph is sparse
• Because $|E|$ is not always near to $|V|^2$ we do not typically substitute $|V|^2$ for $|E|$ in running times, but leave it as a separate variable
Definition: Tree

A Graph $G = (V, E)$ is a tree if it is undirected, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the “root”.
Breadth-First Search

• Input: a node $s$
• Behavior: Start with node $s$, visit all neighbors of $s$, then all neighbors of neighbors of $s$, ...
• Output:
  • How long is the shortest path?
  • Is the graph connected?
void bfs(graph, s)
{
    found = new Queue();
    found.enqueue(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.dequeue();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.enqueue(v);
            }
        }
    }
}
int shortestPath(graph, s, t){
    found = new Queue();
    layer = 0;
    found.enqueue(s);
    mark s as "visited";
    While ( !found.isEmpty() ){
        current = found.dequeue();
        layer = depth of current;
        for ( v : neighbors(current) ){
            if ( !v marked "visited" ){
                mark v as "visited";
                depth of v = layer + 1;
                found.enqueue(v);
            }
        }
    }
    return depth of t;
}
Depth-First Search
Depth-First Search

• Input: a node $s$

• Behavior: Start with node $s$, visit one neighbor of $s$, then all nodes reachable from that neighbor of $s$, then another neighbor of $s$, ...

• Output:
  • Does the graph have a cycle?
  • A topological sort of the graph.
DFS (non-recursive)

void dfs(graph, s){
    found = new Stack();
    found.pop(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.pop();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.push(v);
            }
        }
    }
}

Running time: $\Theta(|V| + |E|)$
DFS Recursively (more common)

```c
void dfs(graph, curr){
    mark curr as “visited”;  
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;  
}
```
Using DFS

- Consider the “visited times” and “done times”
- Edges can be categorized:
  - **Tree Edge**
    - \((a, b)\) was followed when pushing
    - \((a, b)\) when \(b\) was unvisited when we were at \(a\)
  - **Back Edge**
    - \((a, b)\) goes to an “ancestor”
    - \(a\) and \(b\) visited but not done when we saw \((a, b)\)
    - \(t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)\)
  - **Forward Edge**
    - \((a, b)\) goes to a “descendent”
    - \(b\) was visited and done between when \(a\) was visited and done
    - \(t_{visited}(a) < t_{visited}(b) < t_{done}(b) < t_{done}(a)\)
  - **Cross Edge**
    - \((a, b)\) goes to a node that doesn’t connect to \(a\)
    - \(b\) was seen and done before \(a\) was ever visited
    - \(t_{done}(b) < t_{visited}(a)\)
Cycle Detection

```java
boolean hasCycle(graph, curr){
    mark curr as “visited”;
    cycleFound = false;
    for (v : neighbors(current)){
        if (v marked “visited” && !v marked “done”){
            cycleFound=true;
        }
        if (!v marked “visited” && !cycleFound){
            cycleFound = hasCycle(graph, v);
        }
    }
    mark curr as “done”;
    return cycleFound;
}
```

Idea: Look for a back edge!
A Topological Sort of a **directed acyclic graph** $G = (V, E)$ is a permutation of $V$ such that if $(u, v) \in E$ then $u$ is before $v$ in the permutation.
Topological Sort

Idea: List in descending order by “done” time

```java
List topologicalSort(graph){
    doneList = new List();
    for (v : graph.vertices()){
        if (! v marked as “seen”){
            topSortRec(graph, v, doneList);
        }
    }
    doneList.reverse();
    return doneList;
}

void topSortRec(graph, curr, doneList){
    mark curr as “visited”;  
    for (v : neighbors(current)){
        if (! v marked “visited”){
            topSortRec(graph, v);
        }
    }
    mark curr as “done”;  
    doneList.add(curr);
}```