Graph Algorithms: Shortest Paths

CS211
Fall 2000

There are several sorting methods that take linear time:
- Counting Sort
  - Sorts integers from a small range: \([0..k]\)
  - \(k = O(n)\)
- Radix Sort
  - The method used by the old card-sorters
  - Sorting time \(O(dn)\)
  - \(d\) is the number of "digits"

How do these methods get around the \(\Omega(n \log n)\) lower bound?
- They don’t use comparisons

What sorting method works best?
- QuickSort is best general-purpose sort
- Counting Sort or Radix Sort can be best for some kinds of data

Aside: An Open Question on Sorting

How long does it take to sort an \(n\)-by-\(n\) table of numbers?

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>15</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>22</td>
</tr>
</tbody>
</table>

\(O(n^2 \log n)\) because there are \(n^2\) numbers in the table
- Shouldn’t it be easier to sort than an arbitrary set of \(n^2\) numbers?

Recall Digraphs

- **Adjacency Matrix**
  - Space \(O(v^2)\)
  - \(g[u][v]\) is true if there is an edge from \(u\) to \(v\)

- **Adjacency List**
  - Space \(O(e + v)\)
  - The list for \(u\) contains \(v, c\) if there is an edge from \(u\) to \(v\) with cost \(c\)

Recall Weighted Digraphs

- **Adjacency Matrix**
  - \(g[u][v]\) is \(c\) if there is an edge of cost \(c\) from \(u\) to \(v\)

- **Adjacency List**
  - The list for \(u\) contains \(v, c\) if there is an edge from \(u\) to \(v\) that has cost \(c\)

Goal: Find Shortest Path in a Graph

- Finding the shortest (min-cost) path in a graph is a problem that occurs often
- Find the least-cost route between Ithaca and Detroit
- Result depend on our notion of cost
  - Least mileage
  - Least time
  - Cheapest
  - Least boring
- All of these "costs" can be represented as edge costs on a graph
- How do we find a shortest path?
Shortest Paths for Unweighted Graphs

bfsDistance(s):
// s is the start vertex
// dist[v] is length of s-to-v path
// Initially dist[v] = \infty for all v
Q.insert(s);
while (Q nonempty) {
    v = Q.get();
    for (each w adjacent to v) {
        if (dist[w] == \infty) {
            dist[w] = dist[v]+1;
            Q.insert(w);
        }  // if
    }  // for
}  // while

Analysis for bfsDistance

How many times can a vertex be placed in the queue?
How much time for the for-loop?
Depends on representation
- Adjacency Matrix: O(v)
- Adjacency List: O(e + v)

Time:
- O(v^2) for adj matrix
- O(e + v) for adj list

If There are Edge Costs?

Idea #1
- Add false nodes so that all edge costs are 1
- But what if edge costs are large?
- What if the costs aren't integers?

Idea #2
- Nothing "interesting" happens at the false nodes
- Can't we just jump ahead to the next "real" node
- Rule: always do the closest (real) node first
- Use the array dist[] to:
  - Report answers
  - Keep track of what to do next

Dijkstra's Algorithm

Intuition
- Edges are threads; vertices are beads
- Pick up at s; mark each node as it leave the table
Note: Negative edge-costs are not allowed

dijkstra(s):
    dist[s] = 0;
while (some vertices are unmarked) {
    v = unmarked vertex with smallest dist;
    Mark v;
    for (each w adj to v) {
        dist[w] = min [ dist[w], dist[v] + c(v,w) ];
    }  // for
}  // while

Dijkstra's Algorithm using Adj Matrix

While-loop is done v times
Within the loop
- Choosing v takes O(v)
- For-loop takes O(v) time
Total time = O(v^2)

Dijkstra's Algorithm using Adj List

Looks like we need a PQ
Problem: priorities are updated as algorithm runs
Can insert pair (v, dist[v]) in PQ whenever dist[v] is updated
At most e things in PQ
Time O(e + e log e)
Using a more complicated PQ (e.g., Pairing Heap), time can be brought down to O(e + v log v)