Lecture 20: Topo-Sort and Dijkstra’s Greedy Idea

✦ Items on Today’s Lunch Menu:
  ✦ Topological Sort (ver. 1 & 2): Gunning for linear time…
  ✦ Finding Shortest Paths
    ♦ Breadth-First Search
    ♦ Dijkstra’s Method: Greed is good!

✦ Covered in Chapter 9 in the textbook

---

Graph Algorithm #1: Topological Sort

**Problem:** Find an order in which all these courses can be taken.

Example: 142 143 378
370 321 341 322
326 421 401

R. Rao, CSE 326

Some slides based on: CSE 326 by S. Wolfman, 2000
Topological Sort Definition

**Topological sorting problem**: given digraph $G = (V, E)$, find a linear ordering of vertices such that:
for all edges $(v, w)$ in $E$, $v$ precedes $w$ in the ordering

Any linear ordering in which all the arrows go to the right is a valid solution
Topological Sort

Topological sorting problem: given digraph $G = (V, E)$, find a linear ordering of vertices such that:
for any edge $(v, w)$ in $E$, $v$ precedes $w$ in the ordering

Step 1: Identify vertices that have no incoming edge
- The “in-degree” of these vertices is zero
Step 1: Identify vertices that have no incoming edge
- If no such edges, graph has cycles (cyclic graph)

Example of a cyclic graph:
No vertex of in-degree 0

R. Rao, CSE 326
Topological Sort Algorithm

**Step 2**: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.

Repeat **Steps 1** and **Step 2** until graph is empty
Topological Sort Algorithm

Repeat Steps 1 and Step 2 until graph is empty
Topological Sort Algorithm

Repeat Steps 1 and Step 2 until graph is empty

Final Result:

Summary of Topo-Sort Algorithm #1

1. Store each vertex’s In-Degree (# of incoming edges) in an array
2. While there are vertices remaining:
   - Find a vertex with In-Degree zero and output it
   - Reduce In-Degree of all vertices adjacent to it by 1
   - Mark this vertex (In-Degree = -1)
Topological Sort Algorithm #1: Analysis

For input graph \( G = (V, E) \), Run Time = ?

\textit{Break down into total time required to:}

§ **Initialize In-Degree array:**
\[ O(|E|) \]

§ **Find vertex with in-degree 0:**
\[ |V| \text{ vertices, each takes } O(|V|) \text{ to search In-Degree array.} \]
\[ \text{Total time} = O(|V|^2) \]

§ **Reduce In-Degree of all vertices adjacent to a vertex:**
\[ O(|E|) \]

§ **Output and mark vertex:**
\[ O(|V|) \]

\[ \text{Total time} = O(|V|^2 + |E|) \quad \text{Quadratic time!} \]

Can we do better than quadratic time?

\textbf{Problem:}

Need a faster way to find vertices with in-degree 0 instead of searching through entire in-degree array
Topological Sort (Take 2)

**Key idea:** Initialize and maintain a queue (or stack) of vertices with In-Degree 0

**Queue:**

```
A  F
```

In-Degree array:

```
A  B  C  D  E  F
0  1  1  2  2  0
```

Adjacency list:

```
A —— B —— D
B —— C
C —— D
D —— E
E —— F
F
```

Topological Sort (Take 2)

After each vertex is output, when updating In-Degree array, enqueue any vertex whose In-Degree has become zero

```
A dequeue
B enqueue
```

```
A  B  C  D  E  F
0  0  1  1  2  0
```

Output:

```
A
```

**Adjacency list:**

```
A —— B —— D
B —— C
C —— D
D —— E
E —— F
F
```
Topological Sort Algorithm #2

1. Store each vertex’s **In-Degree** in an array
2. Initialize a queue with all in-degree zero vertices
3. While there are vertices remaining in the queue:
   - Dequeue and output a vertex
   - Reduce In-Degree of all vertices adjacent to it by 1
   - Enqueue any of these vertices whose In-Degree became zero

Sort this digraph!

Topological Sort Algorithm #2: Analysis

For input graph $G = (V,E)$, Run Time = ?

*Break down into total time to:*
  - Initialize In-Degree array:
    - $O(|E|)$
  - Initialize Queue with In-Degree 0 vertices:
    - $O(|V|)$
  - Dequeue and output vertex:
    - $|V|$ vertices, each takes only $O(1)$ to dequeue and output.
    - Total time = $O(|V|)$
  - Reduce In-Degree of all vertices adjacent to a vertex and
  - Enqueue any In-Degree 0 vertices:
    - $O(|E|)$

*Total time* = $O(|V| + |E|)$  **Linear running time!**
Paths

- Recall definition of a path in a tree – same for graphs
- A path is a list of vertices \( \{v_1, v_2, \ldots, v_n\} \) such that \( (v_i, v_{i+1}) \) is in \( E \) for all \( 0 \leq i < n \).

Example of a path:

\[ p = \{Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle\} \]

Simple Paths and Cycles

- A simple path repeats no vertices (except the 1st can be the last):
  - \( p = \{Seattle, Salt Lake City, San Francisco, Dallas\} \)
  - \( p = \{Seattle, Salt Lake City, Dallas, San Francisco, Seattle\} \)
- A cycle is a path that starts and ends at the same node:
  - \( p = \{Seattle, Salt Lake City, Dallas, San Francisco, Seattle\} \)
- A simple cycle is a cycle that repeats no vertices except that the first vertex is also the last
- A directed graph with no cycles is called a DAG (directed acyclic graph) E.g. All trees are DAGs
  - A graph with cycles is often a drag…
Path Length and Cost

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge
  - Note: Path length = unweighted path cost (edge weight = 1)

Seattle

San Francisco

Dallas

Salt Lake City

Chicago

\[ \text{length}(p) = 5 \]
\[ \text{cost}(p) = 11.5 \]

Single Source, Shortest Path Problems

- Given a graph \( G = (V, E) \) and a “source” vertex \( s \) in \( V \), find the **minimum cost paths** from \( s \) to every vertex in \( V \)
- **Many variations**:
  - unweighted vs. weighted
  - cyclic vs. acyclic
  - positive weights only vs. negative weights allowed
  - multiple weight types to optimize
  - Etc.
- We will look at only a couple of these...
  - See text for the others
Why study shortest path problems?

✦ Plenty of applications
✦ Traveling on a “starving student” budget: What is the cheapest multi-stop airline schedule from Seattle to city X?
✦ Optimizing routing of packets on the internet:
  ➤ Vertices = routers, edges = network links with different delays
  ➤ What is the routing path with smallest total delay?
✦ Hassle-free commuting: Finding what highways and roads to take to minimize total delay due to traffic
✦ Finding the fastest way to get to coffee vendors on campus from your classrooms

Unweighted Shortest Paths Problem

**Problem:** Given a “source” vertex $s$ in an unweighted graph $G = (V,E)$, find the shortest path from $s$ to all vertices in $G$

Find the shortest path from $C$ to: A B C D E F G H
Solution based on **Breadth-First Search**

- **Basic Idea:** Starting at node \( s \), find vertices that can be reached using 0, 1, 2, 3, ..., \( N-1 \) edges (works even for cyclic graphs!)

On-board example:
Find the shortest path from \( C \) to: \( A \ B \ C \ D \ E \ F \ G \ H \)

---

**Breadth-First Search (BFS) Algorithm**

- Uses a **queue** to store vertices that need to be expanded

- Pseudocode (source vertex is \( s \)):
  1. \( \text{Dist}[s] = 0 \)
  2. Enqueue(s)
  3. While queue is not empty
     1. \( X = \text{dequeue} \)
     2. For each vertex \( Y \) adjacent to \( X \) and **not previously visited**
        - \( \text{Dist}[Y] = \text{Dist}[X] + 1 \)
        - \( \text{Prev}[Y] = X \)
        - Enqueue \( Y \)

- Running time (same as topological sort) = \( O(|V| + |E|) \) (why?)
That was easy but what if edges have weights?

Does BFS still work for finding minimum cost paths?

Can you find a counterexample (a path) for this graph to show BFS won’t work?

What if edges have weights?

- BFS does not work anymore – minimum cost path may have additional hops

**Shortest path from C to A:**

**BFS:** C → A

(cost = 9)

**Minimum Cost Path:** C → E → D → A

(cost = 8)
Dijkstra to the rescue…

✦ Legendary figure in computer science
✦ Some rumors collected from previous classes…
✦ Rumor #1: Supported teaching introductory computer courses without computers (pencil and paper programming)
✦ Rumor #2: Supposedly wouldn’t read his e-mail; so, his staff had to print out his e-mails and put them in his mailbox

An Aside: Dijsktra on GOTOs

“For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce.”

Dijkstra’s Algorithm for Weighted Shortest Path

✦ Classic algorithm for solving shortest path in weighted graphs (without negative weights)

✦ Example of a greedy algorithm
  ➔ Irrevocably makes decisions without considering future consequences
  ➔ Sound familiar? Not necessarily the best life strategy…
    but works in some cases (e.g. Huffman encoding)

---

Dijkstra’s Algorithm for Weighted Shortest Path

✦ Basic Idea:
  ➔ Similar to BFS
    ♦ Each vertex stores a cost for path from source
    ♦ Vertex to be expanded is the one with least path cost seen so far
      ♦ Greedy choice – always select current best vertex
      ♦ Update costs of all neighbors of selected vertex
  ➔ But unlike BFS, a vertex already visited may be updated if a better path to it is found
Pseudocode for Dijkstra’s Algorithm

1. Initialize the cost of each node to $\infty$
2. Initialize the cost of the source to 0
3. While there are unknown nodes left in the graph
   1. Select the unknown node $N$ with the lowest cost (greedy choice)
   2. Mark $N$ as known
3. For each node $X$ adjacent to $N$
   If $(N$’s cost + cost of $(N, X)) < X$’s cost
   $X$’s cost = $N$’s cost + cost of $(N, X)$
   $\text{Prev}[X] = N$ //store preceding node

Dijkstra’s Algorithm (greed in action)

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initial | Final

(Prev allows paths to be reconstructed)
Dijkstra’s Algorithm (greed in action)

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>∞</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vertex</th>
<th>known</th>
<th>cost</th>
<th>Prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
<td>2</td>
<td>C</td>
</tr>
</tbody>
</table>

Initial

Final

Questions for Next Time:
Does Dijkstra’s method always work?
How fast does it run?
Where else in life can I be greedy?

To Do:
Start Homework Assignment #4
(Don’t wait until the last few days!!!)
Continue reading and enjoying chapter 9