CSE 373
Data Structures & Algorithms

Lectures 19-20
Graphs
Graph... ADT?

- Not quite an ADT... operations not clear

- A formalism for representing relationships between objects
  
  Graph $G = (V, E)$
  - Set of vertices:
    $V = \{v_1, v_2, \ldots, v_n\}$
  - Set of edges:
    $E = \{e_1, e_2, \ldots, e_m\}$
    where each $e_i$ connects two vertices $(v_{i1}, v_{i2})$

  $V = \{\text{Han, Leia, Luke}\}$
  $E = \{(\text{Luke, Leia}), (\text{Han, Leia}), (\text{Leia, Han})\}$
Examples of Graphs

• The web
  – Vertices are webpages
  – Each edge is a link from one page to another

• Call graph of a program
  – Vertices are subroutines
  – Edges are calls and returns

• Social networks
  – Vertices are people
  – Edges connect friends
Graph Definitions

In *directed* graphs, edges have a direction:

In *undirected* graphs, they don’t (are two-way):

\[ v \text{ is adjacent to } u \text{ if } (u, v) \in E \]
Weighted Graphs

Each edge has an associated weight or cost.
Paths and Cycles

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

• A cycle is a path that begins and ends at the same node.

\[ p = \{Seattle, SaltLakeCity, Chicago, Dallas, SanFrancisco, Seattle\} \]
Path Length and Cost

- **Path length**: the number of edges in the path
- **Path cost**: the sum of the costs of each edge

\[
\text{length}(p) = 5 \quad \text{cost}(p) = 11.5
\]
More Definitions:
Simple Paths and Cycles

A **simple path** repeats no vertices (except that the first can also be the last):
- \( p = \{ \text{Seattle, Salt Lake City, San Francisco, Dallas} \} \)
- \( p = \{ \text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle} \} \)

A **cycle** is a path that starts and ends at the same node:
- \( p = \{ \text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle} \} \)
- \( p = \{ \text{Seattle, Salt Lake City, Seattle, San Francisco, Seattle} \} \)

A **simple cycle** is a cycle that is also a simple path (in undirected graphs, no edge can be repeated)
Trees as Graphs

• Every tree is a graph with some restrictions:
  – the tree is directed
  – there is exactly one directed path from the root to every node
Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

Aside: If program call-graph is a DAG, then all procedure calls can be in-lined

\{Tree\} \subset \{DAG\} \subset \{Graph\}
Rep 1: Adjacency Matrix

A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$.

Runtimes:
- Iterate over vertices?
- Iterate over edges?
- Iterate edges adj. to vertex?
- Existence of edge?

Space requirements?
Rep 2: Adjacency List

A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices.

Runtimes:
- Iterate over vertices?
- Iterate over edges?
- Iterate edges adj. to vertex?
- Existence of edge?

Space requirements?
Some Applications:
Moving Around Washington

What’s the *shortest way* to get from Seattle to Pullman?

Edge labels:

Distance
Some Applications: Moving Around Washington

What’s the *fastest way* to get from Seattle to Pullman?

Edge labels: Distance, speed limit

11/23/2009 CSE 373 Fall 2009 -- Dan Suciu
Some Applications: Reliability of Communication

If Wenatchee’s phone exchange *goes down*, can Seattle still talk to Pullman?
Some Applications:
Bus Routes in Downtown Seattle

If we’re at 3\textsuperscript{rd} and Pine, how can we get to 1\textsuperscript{st} and University using Metro? How about 4\textsuperscript{th} and Seneca?
Graph Connectivity

- Undirected graphs are *connected* if there is a path between any two vertices.

- Directed graphs are *strongly connected* if there is a path from any one vertex to any other.

- Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*.

- A *complete* graph has an edge between every pair of vertices.
Graph Traversals

- Breadth-first search (and depth-first search) work for arbitrary (directed or undirected) graphs - not just mazes!
  - Must mark visited vertices. Why?
  - So you do not go into an infinite loop! It’s not a tree.
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly/strongly) connected?
- Which one:
  - Uses a queue?
  - Uses a stack?
  - Always finds the shortest path (for unweighted graphs)?
The Shortest Path Problem

• Given a graph $G$, edge costs $c_{i,j}$, and vertices $s$ and $t$ in $G$, find the shortest path from $s$ to $t$.

• For a path $p = v_0 \ v_1 \ v_2 \ldots \ v_k$
  - *unweighted length* of path $p = k$ (a.k.a. *length*)

  - *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (a.k.a *cost*)

  — Path length equals path cost when ?
Single Source Shortest Paths (SSSP)

• Given a graph $G$, edge costs $c_{i,j}$, and vertex $s$, find the shortest paths from $s$ to all vertices in $G$.

  – Is this harder or easier than the previous problem?
All Pairs Shortest Paths (APSP)

- Given a graph $G$ and edge costs $c_{i,j}$, find the shortest paths between all pairs of vertices in $G$.

  - Is this harder or easier than SSSP?

  - Could we use SSSP as a subroutine to solve this?
Breadth-First Graph Search

BFS( Start)
for all nodes x do x.dist = ∞;
Start.dist = 0;
enqueue(Start, Open);
repeat
  if (empty(Open)) then return;
x:= dequeue(Open);
  for each y in children(x) do
    if (y.dist = ∞)
      then { y.dist = x.dist + 1;
               enqueue(y, Open); }
Depth-First Graph Search

DFS( Start)
for all nodes x do x.dist = ∞;
Start.dist = 0;
push(Start, Open);
repeat
  if (empty(Open)) then return;
x:= pop(Open);
  for each y in children(x) do
    if (y.dist > x.dist + 1)
      then { y.dist = x.dist + 1;
             push(y, Open); }
  end-repeat
Comparison: DFS versus BFS

• Depth-first search
  – Does not find shortest paths naturally
    • Had to do the extra test $y.\text{dist} > x.\text{dist} + 1$
  – Must be careful to mark visited vertices (using $x.\text{dist}$, or some other means), or you could go into an infinite loop if there is a cycle

• Breadth-first search
  – Always finds shortest paths – optimal solutions
  – Marking visited nodes can improve efficiency, but even without doing so search is guaranteed to terminate

  – Is BFS always preferable?
DFS Space Requirements

• Assume:
  – Longest path in graph is length $d$
  – Highest number of out-edges is $k$

• DFS stack grows at most to size $dk$
  – For $k=10$, $d=15$, size is 150
BFS Space Requirements

- Assume
  - Distance from start to a goal is $d$
  - Highest number of out edges is $k$ BFS
- Queue could grow to size $k^d$
  - For $k=10$, $d=15$, size is 1,000,000,000,000,000
Conclusion

• For large graphs, DFS is more memory efficient, if we can limit the maximum path length to some fixed $d$.
  
  – If we knew the distance from the start to the goal in advance, we can just not add any children to stack after level $d$
  
  – But what if we don’t know $d$ in advance?
Edsger Wybe Dijkstra
(1930-2002)

• Invented concepts of structured programming, synchronization, weakest precondition, and "semaphores" for controlling computer processes. The Oxford English Dictionary cites his use of the words "vector" and "stack" in a computing context.

• Believed programming should be taught without computers

• 1972 Turing Award

• “In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”
Shortest Path for Weighted Graphs

• Given a graph \( G = (V, E) \) with edge costs \( c(e) \), and a vertex \( s \in V \), find the shortest (lowest cost) path from \( s \) to every vertex in \( V \)

• Assume: only positive edge costs
Dijkstra’s Algorithm for Single Source Shortest Path

• Similar to breadth-first search, but uses a heap instead of a queue:
  – Always select (expand) the vertex that has a lowest-cost path to the start vertex

• Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges
Dijkstra’s Algorithm: Idea

Adapt BFS to handle weighted graphs

Two kinds of vertices:

- Finished or known vertices
  - Shortest distance has been computed
  - Have tentative distance
- Unknown vertices
Dijkstra’s Algorithm: Idea

At each step:
1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances
Dijkstra’s Algorithm: Pseudocode

Initialize the cost of each node to $\infty$

Initialize the cost of the source to 0

While there are unknown nodes left in the graph
   Select an unknown node $b$ with the lowest cost
   Mark $b$ as known
   For each node $a$ adjacent to $b$
      if $b$’s cost + cost of $(b, a) < a$’s old cost
         $a$’s cost = $b$’s cost + cost of $(b, a)$
         $a$’s prev path node = $b$
Important Features

• Once a vertex is made known, the cost of the shortest path to that node is known

• While a vertex is still not known, another shorter path to it might still be found

• The shortest path itself can be found by following the backward pointers stored in node.path
Dijkstra’s Algorithm in action

![Graph Diagram]

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>0</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>
<tr>
<td>F</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm in action

Vertex | Visited? | Cost | Found by
--- | --- | --- | ---
A | Y | 0 | A
B | <=2 | A
C | <=1 | A
D | <=4 | A
E | ?? | |
F | ?? | |
G | ?? | |
H | ?? | |
Dijkstra’s Algorithm in action

Here is a visual representation of Dijkstra's Algorithm in action, along with a table summarizing the visited vertices, their costs, and the vertices that found them.

- **Vertex**: A, B, C, D, E, F, G, H
- **Visited?**: Y (yes), <= (less than or equal to)
- **Cost**: 0, <=2, 1, <=4, <=12, ??
- **Found by**: A, A, A, A, C, ??, ??, ??

The diagram shows the network of vertices with weights on the edges, and the table tracks the progress of the algorithm as vertices are visited and the shortest path costs are updated.
Dijkstra’s Algorithm in action

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>&lt;=4</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>&lt;=12</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>&lt;=4</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>G</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm in action

![Diagram of Dijkstra's Algorithm]

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>&lt;=12</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>&lt;=4</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>??</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>??</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11/23/2009  CSE-373 Fall 2009 -- Dan Suciu
Dijkstra’s Algorithm in action

Vertex | Visited? | Cost | Found by
--- | --- | --- | ---
A | Y | 0 |
B | Y | 2 | A
C | Y | 1 | A
D | Y | 4 | A
E | <=12 | C |
F | Y | 4 | B
G | ?? | |
H | <=7 | F |
Dijkstra’s Algorithm in action

Vertex | Visited? | Cost | Found by
--- | --- | --- | ---
A | Y | 0 | A
B | Y | 2 | A
C | Y | 1 | A
D | Y | 4 | A
E | <=12 | C
F | Y | 4 | B
G | <=8 | H
H | Y | 7 | F
Dijkstra’s Algorithm in action

Vertex | Visited? | Cost | Found by
-------|----------|------|----------
A       | Y        | 0    |          
B       | Y        | 2    | A        
C       | Y        | 1    | A        
D       | Y        | 4    | A        
E       | <=11     |      | G        
F       | Y        | 4    | B        
G       | Y        | 8    | H        
H       | Y        | 7    | F        

11/23/2009
Dijkstra’s Algorithm in action

Vertex Visited? Cost Found by
A Y 0 A
B Y 2 A
C Y 1 A
D Y 4 A
E Y 11 G
F Y 4 B
G Y 8 H
H Y 7 F
Another

\begin{center}
\begin{tabular}{|l|c|c|c|}
\hline
\textbf{V} & \textbf{Visited?} & \textbf{Cost} & \textbf{Found by} \\
\hline
v0 & & & \\
v1 & & & \\
v2 & & & \\
v3 & & & \\
v4 & & & \\
v5 & & & \\
v6 & & & \\
\hline
\end{tabular}
\end{center}
Another

<table>
<thead>
<tr>
<th>V</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>&lt;= 2</td>
<td>V0</td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td>&lt;= 1</td>
<td>V0</td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Another

\[
\begin{array}{|c|c|c|c|}
\hline
V & Visited? & Cost & Found by \\
\hline
v0 & Y & 0 & \\
\hline
v1 & <= 6 & V3 & \\
\hline
v2 & <= 2 & V0 & \\
\hline
v3 & Y & 1 & V0 \\
\hline
v4 & <= 2 & V3 & \\
\hline
v5 & <= 7 & V3 & \\
\hline
v6 & <= 6 & V3 & \\
\hline
\end{array}
\]
Another

V | Visited? | Cost | Found by
---|---------|------|---------
v0 | Y       | 0    |         
v1 |         | <= 6 | V3      
v2 | Y       | 2    | V0      
v3 | Y       | 1    | V0      
v4 |         | <= 2 | V3      
v5 |         | <= 4 | V2      
v6 |         | <= 6 | V3      

s

V0 –> 2 –> V2 –> 1 –> V3 –> 1 –> V1 –> 5 –> V5 –> 10 –> V6 –> 3
Another

\begin{itemize}
  \item $s$
  \item $v_0$
  \item $v_1$
  \item $v_2$
  \item $v_3$
  \item $v_4$
  \item $v_5$
  \item $v_6$
\end{itemize}

\begin{tabular}{|c|c|c|c|}
\hline
$V$ & Visited? & Cost & Found by \\
\hline
$v_0$ & Y & 0 & \\
\hline
$v_1$ & & <= 3 & V4 \\
\hline
$v_2$ & Y & 2 & V0 \\
\hline
$v_3$ & Y & 1 & V0 \\
\hline
$v_4$ & Y & 2 & V3 \\
\hline
$v_5$ & & <= 4 & V2 \\
\hline
$v_6$ & & <= 6 & V3 \\
\hline
\end{tabular}
Another

<table>
<thead>
<tr>
<th>V</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>Y</td>
<td>3</td>
<td>V4</td>
</tr>
<tr>
<td>v2</td>
<td>Y</td>
<td>2</td>
<td>V0</td>
</tr>
<tr>
<td>v3</td>
<td>Y</td>
<td>1</td>
<td>V0</td>
</tr>
<tr>
<td>v4</td>
<td>Y</td>
<td>2</td>
<td>V3</td>
</tr>
<tr>
<td>v5</td>
<td>&lt;= 4</td>
<td></td>
<td>V2</td>
</tr>
<tr>
<td>v6</td>
<td>&lt;= 6</td>
<td></td>
<td>V3</td>
</tr>
</tbody>
</table>
Another

<table>
<thead>
<tr>
<th>V</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>Y</td>
<td>3</td>
<td>V4</td>
</tr>
<tr>
<td>v2</td>
<td>Y</td>
<td>2</td>
<td>V0</td>
</tr>
<tr>
<td>v3</td>
<td>Y</td>
<td>1</td>
<td>V0</td>
</tr>
<tr>
<td>v4</td>
<td>Y</td>
<td>2</td>
<td>V3</td>
</tr>
<tr>
<td>v5</td>
<td>Y</td>
<td>4</td>
<td>V2</td>
</tr>
<tr>
<td>v6</td>
<td>&lt;= 6</td>
<td></td>
<td>V3</td>
</tr>
</tbody>
</table>
Another

<table>
<thead>
<tr>
<th>V</th>
<th>Visited?</th>
<th>Cost</th>
<th>Found by</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>Y</td>
<td>3</td>
<td>v4</td>
</tr>
<tr>
<td>v2</td>
<td>Y</td>
<td>2</td>
<td>v0</td>
</tr>
<tr>
<td>v3</td>
<td>Y</td>
<td>1</td>
<td>v0</td>
</tr>
<tr>
<td>v4</td>
<td>Y</td>
<td>2</td>
<td>v3</td>
</tr>
<tr>
<td>v5</td>
<td>Y</td>
<td>4</td>
<td>v2</td>
</tr>
<tr>
<td>v6</td>
<td>Y</td>
<td>6</td>
<td>v3</td>
</tr>
</tbody>
</table>
```cpp
void Graph::dijkstra(Vertex s) {
    Vertex v, w;

    Initialize s.dist = 0 and set dist of all other vertices to infinity

    while (there exist unknown vertices, find the one \( b \) with the smallest distance)
        \( b \).known = true;

        for each a adjacent to \( b \)
            if (!a.known)
                if (b.dist + weight(b, a) < a.dist)
                    a.dist = (b.dist + weight(b, a));
                    a.path = b;

    deleteMin on a heap…
}
```

Running time: \( O(|E| \log |V|) \) – there are \( |E| \) edges to examine, and each one causes a heap operation of time \( O(\log |V|) \)
Dijkstra’s Algorithm: Summary

• Classic algorithm for solving SSSP in weighted graphs *without negative weights*

• A *greedy* algorithm (irrevocably makes decisions without considering future consequences)

• Intuition for correctness:
  – shortest path from source vertex to itself is 0
  – cost of going to adjacent nodes is at most edge weights
  – cheapest of these must be shortest path to that node
  – update paths for new node and continue picking cheapest path
How does Dijkstra’s decide which vertex to add to the Known set next?

- If path to $V$ is shortest, path to $W$ must be \textit{at least as long} (or else we would have picked $W$ as the next vertex)
- So the path through $W$ to $V$ cannot be any shorter!
Correctness: Inside the Cloud

Prove by induction on # of nodes in the cloud:
Initial cloud is just the source with shortest path 0

**Assume**: Everything inside the cloud has the correct shortest path

**Inductive step**: Only when we prove the shortest path to some node \( v \) (which is not in the cloud) is correct, we add it to the cloud

When does Dijkstra’s algorithm not work?
The Trouble with Negative Weight Cycles

What’s the shortest path from A to E? Problem?
Dijkstra’s vs BFS

At each step:

1) Pick closest unknown vertex
2) Add it to finished vertices
3) Update distances

Dijkstra’s Algorithm

At each step:

1) Pick vertex from queue
2) Add it to visited vertices
3) Update queue with neighbors

Breadth-first Search
Two Questions

- What if I had multiple potential start points, and need to know the minimum cost of reaching each node from any start point?

- What if I want to know the minimum cost between every pair of nodes in the graph?
Single-Source Shortest Path

• Given a graph $G = (V, E)$ and a single distinguished vertex $s$, find the shortest weighted path from $s$ to every other vertex in $G$.

All-Pairs Shortest Path:

• Find the shortest paths between all pairs of vertices in the graph.

• How?
Analysis

• Total running time for Dijkstra’s:
  \[ O(|V| \log |V| + |E| \log |V|) \] (heaps)

  What if we want to find the shortest path from each point to ALL other points?
Dynamic Programming

Algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results (rather than re-computing them).

**Simple Example**: Calculating the Nth Fibonacci number.

\[ \text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2) \]
Floyd-Warshall

\[
\begin{aligned}
&\text{for (int } k = 1; k \leq V; k++) \\
&\text{ for (int } i = 1; i \leq V; i++) \\
&\text{ for (int } j = 1; j \leq V; j++) \\
&\quad \text{if ( ( } M[i][k] + M[k][j] \text{ ) } < M[i][j] \text{ ) } \\
&\quad M[i][j] = M[i][k] + M[k][j]
\end{aligned}
\]

**Invariant:** After the kth iteration, the matrix includes the shortest paths for all pairs of vertices (i,j) containing only vertices 1..k as intermediate vertices.
Initial state of the matrix:

\[
\begin{array}{ccccc}
  & a & b & c & d & e \\
a & 0 & 2 & - & -4 & - \\
b & - & 0 & -2 & 1 & 3 \\
c & - & - & 0 & - & 1 \\
d & - & - & - & 0 & 4 \\
e & - & - & - & - & 0 \\
\end{array}
\]

\[M[i][j] = \min(M[i][j], M[i][k] + M[k][j])\]
Floyd-Warshall - for All-pairs shortest path

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>0</td>
<td>-2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Final Matrix Contents
This is a partial ordering, for sorting we had a total ordering

Application: Topological Sort

Given a directed graph, $G = (V, E)$, output all the vertices in $V$ such that no vertex is output before any other vertex with an edge to it.

Is the output unique?

Minimize and DO a topo sort
Topological Sort: Take One

1. Label each vertex with its *in-degree* (# of inbound edges)

2. **While** there are vertices remaining:
   a. Choose a vertex $v$ of *in-degree zero*; output $v$
   b. Reduce the in-degree of all vertices adjacent to $v$
   c. Remove $v$ from the list of vertices

*Runtime:*
void Graph::topsort()
{
  Vertex v, w;

  labelEachVertexWithItsIn-degree();

  for (int counter=0; counter < NUM_VERTICES; counter++)
  {
    v = findNewVertexOfDegreeZero();
    v.topologicalNum = counter;
    for each w adjacent to v
      w.indegree--;
  }
}

What's the bottleneck?

O(depends)
Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue $Q$ to contain all in-degree zero vertices
3. While $Q$ not empty
   a. $v = Q$.dequeue; output $v$
   b. Reduce the in-degree of all vertices adjacent to $v$
   c. If new in-degree of any such vertex $u$ is zero
      $Q$.enqueue($u$)

**Note**: could use a stack, list, set, box, … instead of a queue

**Runtime:**
void Graph::topsort()
{
    Queue q(NUM_VERTICES);  int counter = 0; Vertex v, w;
    labelEachVertexWithItsIn-degree();

    q.makeEmpty();
    for each vertex v
        if (v.indegree == 0)
            q.enqueue(v);

    while (!q.isEmpty()){
        v = q.dequeue();
        v.topologicalNum = ++counter;
        for each w adjacent to v
            if (--w.indegree == 0)
                q.enqueue(w);
    }
}

Runtime: $O(|V| + |E|)$