Directed Graph Algorithms

Depth-First Search

Stack (before): A
Stack (after): C

Depth-First Search

Stack (before): C
Stack (after): D, E

Depth-First Search

Stack (before): D (from C), E
Stack (after): D (from C), D (from E)

Depth-First Search

Stack (before): D (from C), D (from E)
Stack (after): D (from C)
Depth-First Search

Stack (before): D (from C)
Stack (after):

- A
- B
- C
- D
- F
- H
- G
- E

discovery edge
cross edge
back edge
forward edge
unexplored edge

Depth-First Search

Stack (before): B
Stack (after):

- A
- C
- B
- D
- F
- H
- G
- E

discovery edge
cross edge
back edge
forward edge
unexplored edge

Depth-First Search

Stack (before): F
Stack (after): H

- A
- B
- C
- D
- F
- H
- G
- E

discovery edge
cross edge
back edge
forward edge
unexplored edge

Depth-First Search

Stack (before): H
Stack (after): G

- A
- B
- C
- D
- F
- H
- G
- E

discovery edge
cross edge
back edge
forward edge
unexplored edge
Depth-First Search

Stack (before): G
Stack (after): I, J, L

Stack (before): I, J, L
Stack (after): I, J

Stack (before): I, J
Stack (after): I, K

Stack (before): I, K
Stack (after): I

Depth-First Search

Breadth-First Search

Queue (before): A
Queue (after): A
Breadth-First Search

Queue (before): A
Queue (after): C

discovery edge
unexplored edge
cross edge
back edge
forward edge

Queue (before): C
Queue (after): D, E

discovery edge
unexplored edge
cross edge
back edge
forward edge

Queue (before): D, E
Queue (after): E

discovery edge
unexplored edge
cross edge
back edge
forward edge

Queue (before): E
Queue (after):

discovery edge
unexplored edge
cross edge
back edge
forward edge

Queue (before): B
Queue (after):

discovery edge
unexplored edge
cross edge
back edge
forward edge
Breadth-First Search

Queue (before): B
Queue (after):

discovery edge
cross edge
back edge
forward edge
unexplored edge

Queue (before): F
Queue (after): H

Queue (before): G
Queue (after): I, J, L

Queue (before): I, J, L
Queue (after): J (from G), L, J (from I)
**Breadth-First Search**

- Queue (before): J (from G), L, J (from I)
- Queue (after): L, J (from I), K

**Topological Sort**

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

- Example: 142 → 143 → 378 → 370 → 321 → 341 → 326 → 421 → 401

In order to take a course, you must take all of its prerequisites first.
Any total ordering in which all the arrows go to the right is a valid solution.

Note that F can go anywhere in this list because it is not connected. Also the solution is not unique.

Any ordering in which an arrow goes to the left is not a valid solution.

NO!

Paths and Cycles

- Given a digraph G = (V,E), a path is a sequence of vertices v₁,v₂,...,vₖ such that:
  - (vᵢ,vᵢ₊₁) in E for 1 ≤ i < k
  - path length = number of edges in the path
  - path cost = sum of costs of each edge
- A path is a cycle if:
  - k > 1; v₁ = vₖ
- G is acyclic if it has no cycles.

Only acyclic graphs can be topologically sorted.

- A directed graph with a cycle cannot be topologically sorted.

Topological sort algorithm: 1

Step 1: Identify vertices that have no incoming edges

- The "in-degree" of these vertices is zero

Topo sort algorithm: 1a

Step 1: Identify vertices that have no incoming edges

- If no such vertices, graph has only cycle(s) (cyclic graph)
- Topological sort not possible – Halt.

Example of a cyclic graph
Topo sort algorithm: 1b

**Step 1:** Identify vertices that have no incoming edges
- Select one such vertex

Select vertices

A
B
C
F
D
E

Topo sort algorithm: 2

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

Select vertices

A
B
C
D
E
F

Continue until done

Repeat **Step 1** and **Step 2** until graph is empty

Select vertices

A
B
C
D
E
F

Select B. Copy to sorted list. Delete B and its edges.

Select vertices

A
B
C
D
E
F

Select C. Copy to sorted list. Delete C and its edges.

Select vertices

A
B
C
D
E
F

Select D. Copy to sorted list. Delete D and its edges.
E, F

Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.

Implementation

Assume adjacency list representation

Translation array

In-Degree array; or add a field to array A

Calculate In-degrees

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

Maintaining Degree 0 Vertices
Topological Sort Algorithm

1. Store each vertex’s In-Degree in an array D
2. Initialize queue with all “in-degree=0” vertices
3. While there are vertices remaining in the queue:
   (a) Dequeue and output a vertex
   (b) Reduce In-Degree of all vertices adjacent to it by 1
   (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.
Topo Sort w/ queue

Queue (before): 6, 2
Queue (after): 2

Answer: 1, 6

Queue (before): 2
Queue (after): 3

Answer: 1, 6, 2

Queue (before): 3
Queue (after): 4

Answer: 1, 6, 2, 3

Queue (before): 4
Queue (after): 5

Answer: 1, 6, 2, 3, 4

Queue (before): 5
Queue (after): 1

Answer: 1, 6, 2, 3, 4, 5

Topo Sort w/ stack

Stack (before): 1
Stack (after): 1, 6

Answer:
Topo Sort w/ stack

Stack (before): 1, 6
Stack (after): 1, 7, 8

Answer: 6

Topo Sort w/ stack

Stack (before): 1, 7, 8
Stack (after): 1, 7

Answer: 6, 8

Topo Sort w/ stack

Stack (before): 1, 7
Stack (after): 1

Answer: 6, 8, 7

Topo Sort w/ stack

Stack (before): 2
Stack (after): 3

Answer: 6, 8, 7, 1, 2

Topo Sort w/ stack

Stack (before): 1
Stack (after): 2

Answer: 6, 8, 7, 1, 2, 3

Topo Sort w/ stack

Stack (before): 3
Stack (after): 4

Answer: 6, 8, 7, 1, 2, 3
Topo Sort w/ stack

Stack (before): 4
Stack (after): 5
Answer: 6, 8, 7, 1, 2, 3, 4

Topo Sort w/ stack

Stack (before): 5
Stack (after): 6
Answer: 6, 8, 7, 1, 2, 3, 4, 5

TopoSort Fails (cycle)

Queue (before): 1
Queue (after): 5
Answer:

TopoSort Fails (cycle)

Queue (before): 1
Queue (after): 2
Answer: 1

TopoSort Fails (cycle)

Queue (before): 2
Queue (after):
Answer: 1, 2

Topological Sort Analysis

- Initialize In-Degree array: $O(|V| + |E|)$
- Initialize Queue with In-Degree 0 vertices: $O(|V|)$
- Dequeue and output vertex:
  - $|V|$ vertices, each takes only $O(1)$ to dequeue and output: $O(|V|)$
- Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices:
  - $O(|E|)$
- For input graph $G=(V,E)$ run time $= O(|V| + |E|)$
  - Linear time!
Recall Path cost, Path length

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

\[
\begin{array}{c}
\text{Seattle} \\
\text{Salt Lake City} \\
\text{San Francisco} \\
\text{Dallas} \\
\text{Chicago}
\end{array}
\]

\[
\begin{array}{cccc}
\text{length(p)} = 5 & 2 & 2 & 3 \\
\text{cost(p)} = 11 & 2 & 2 & 3
\end{array}
\]

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
  - pos. weights only vs. pos. and neg. weights
  - etc.

Why study shortest path problems?

- **Traveling on a budget**: What is the cheapest airline schedule from Seattle to city \( X \)?
- **Optimizing routing of packets on the internet**:
  - Vertices are routers and edges are network links with different delays. What is the routing path with smallest total delay?
- **Shipping**: Find which highways and roads to take to minimize total delay due to traffic
  - etc.

Unweighted Shortest Path

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

\[
\text{Only interested in path lengths}
\]

Breadth-First Search Solution

- **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!)

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{H}
\end{array}
\]

Breadth-First Search Alg.

- **Uses a queue to track vertices that are “nearby”**
- **source vertex is** \( s \)

\[
\begin{array}{l}
\text{Distance}[s] := 0 \\
\text{Enqueue}(Q, s); \text{Mark}(s) \text{//After a vertex is marked once it won’t be enqueued again}
\end{array}
\]

\[
\text{while queue is not empty do}
\]

\[
\begin{array}{l}
\text{X := Dequeue}(Q); \\
\text{for each vertex Y adjacent to X do}
\end{array}
\]

\[
\begin{array}{l}
\text{if Y is unmarked then}
\text{Distance}[Y] := \text{Distance}[X] + 1; \\
\text{Previous}[Y] := X; \text{//if we want to record paths Enqueue(Q, Y); Mark(Y);}
\end{array}
\]

- **Running time = \( O(|V| + |E|) \)**
Example: Shortest Path length

Queue Q = C

Example (ct’d)

Queue Q = A D E

Previous pointer

Indicates the vertex is marked

Example (ct’d)

Q = D E B

Example (ct’d)

Q = B G

Example (ct’d)

Q = F

Example (ct’d)

Q = H
What if edges have weights?

- Breadth First Search does not work anymore
  - minimum cost path may have more edges than minimum length path

Shortest path (length) from C to A:
CÆA (cost = 9)

Minimum Cost Path = CÆEÆDÆA (cost = 8)

Dijkstra’s Algorithm for Weighted Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (without negative weights)
- A greedy algorithm (irrevocably makes decisions without considering future consequences)
- Each vertex has a cost for path from initial vertex

Basic Idea of Dijkstra’s Algorithm

- Find the vertex with smallest cost that has not been “marked” yet.
- Mark it and compute the cost of its neighbors.
- Do this until all vertices are marked.
- Note that each step of the algorithm we are marking one vertex and we won’t change our decision: hence the term “greedy” algorithm