Data Structures in Java


10/18/2015

Daniel Bauer
Today: Graph Traversals

- Depth First Search (a generalization of pre-order traversal on trees to graphs, uses a Stack)
- Breadth First Search (uses a Queue)
- Dijkstra’s algorithm to find weighted shortest paths (uses a Priority Queue)
- Topological sort for Directed Acyclic Graphs.
  - Application: Shortest Project Completion Time.
A **Graph** is a pair of two sets $G=(V,E)$:

- **$V$**: the set of **vertices** (or **nodes**)
- **$E$**: the set of **edges**.
  - each edge is a pair $(v,w)$ where $v,w \in V$

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6 \}$$

$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_5), (v_4, v_6), (v_5, v_6)\}$$
Edges

- Graphs may be **directed** or **undirected**.
  - In directed graphs, the edge pairs are ordered.

- Edges often have some weight or cost associated with them (**weighted** graphs).

\[ V = \{ v_1, v_2, v_3, v_4, v_5, v_6 \} \]
\[ E = \{ (v_1, v_3), (v_2, v_1), (v_2, v_3), (v_3, v_4), (v_3, v_5), (v_4, v_6), (v_5, v_6) \} \]
Paths

- Vertex \( w \) is **adjacent** to vertex \( v \) iff \((w,v) \in E\).
- A **path** is a sequence of vertices \( w_1, w_2, \ldots, w_k \) such that \((w_i, w_{i+1}) \in E\).

**length** of a path:
\[ k - 1 = \text{number of edges on path} \]

**cost** of a path:
\[ \text{Sum of all edge costs.} \]

Path from \( v_1 \) to \( v_6 \), length 3, cost 8:
\((v_1, v_3), (v_3, v_5), (v_5, v_6)\)
Representing Graphs

• Represent graph $G = (E, V)$, option 2: **Adjacency Lists**

• For each vertex, keep a list of all adjacent vertices.
Representing Graphs

- Represent graph $G = (E, V)$, option 2: **Adjacency Lists**
  - For each vertex, keep a list of all adjacent vertices.

Space requirement: $\Theta(|V| + |E|)$
Graph Search
Graph Search
Letters indicate junctions where a decision must be made.
Graph Search
Graph Search
Graph Search: Depth First Search (DFS)

- Goal: Systematically explore the graph, starting at vertex $s$ (source) touching all edges.
- Graph Traversals are the core ingredient of most graph algorithms.

Use a stack.

- Push $s$ to the stack.
- While the stack is not empty:
  - $u \leftarrow$ stack.pop()
  - push all vertices adjacent to $u$ to the stack.
Graph Search: Depth First Search (DFS)

- Goal: Systematically explore the graph, starting at vertex $s$ (source) touching all edges.
- Graph Traversals are the core ingredient of most graph algorithms.

Use a stack:

- Push $s$ to the stack.
- While the stack is not empty:
  - $u \leftarrow$ stack.pop()
  - push all vertices adjacent to $u$ to the stack.
Graph Search: Depth First Search (DFS)

- Goal: Systematically explore the graph, starting at vertex \( s \) (source) touching all edges.
- Graph Traversals are the core ingredient of most graph algorithms.

Use a stack.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - push all vertices adjacent to \( u \) to the stack.
Graph Search: Depth First Search (DFS)

- Goal: Systematically explore the graph, starting at vertex $s$ (source) touching all edges.
- Graph Traversals are the core ingredient of most graph algorithms.

Use a stack.

- Push $s$ to the stack.
- While the stack is not empty:
  - $u \leftarrow$ stack.pop()
  - push all vertices adjacent to $u$ to the stack.
Graph Search: Depth First Search (DFS)

- Goal: Systematically explore the graph, starting at vertex $s$ (source) touching all edges.
- Graph Traversals are the core ingredient of most graph algorithms.

While the stack is not empty:
- $u \leftarrow$ stack.pop()
- push all vertices adjacent to $u$ to the stack.

Problem: This Graph contains cycles!
Depth First Search (DFS) with Visited Set

Use a stack and a set \textit{visited}.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \textit{visited}:
    - add \( u \) to \textit{visited}.
    - push all vertices adjacent to \( u \) to the stack.
Depth First Search (DFS) with Visited Set

Use a stack and a set visited.

- Push s to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack}.\text{pop}() \)
  - if \( u \) is not in visited:
    - add \( u \) to visited.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \{v_1\}
Depth First Search (DFS) with Visited Set

Use a stack and a set visited.

- Push s to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack}.\text{pop}() \)
  - if \( u \) is not in visited:
    - add \( u \) to visited.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \( \{v_1, v_4\} \)
Depth First Search (DFS) with \textit{Visited} Set

Use a stack and a set \textit{visited}.

- Push $s$ to the stack.
- While the stack is not empty:
  - $u \leftarrow \text{stack.pop}()$
  - if $u$ is not in \textit{visited}:
    - add $u$ to \textit{visited}.
    - push all vertices adjacent to $u$ to the stack.

Visited: \{v_1,v_4,v_3\}
Use a stack and a set \textit{visited}.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \textit{visited}:
    - add \( u \) to \textit{visited}.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \( \{v_1, v_4, v_3\} \)
Depth First Search (DFS) with \textit{Visited} Set

Use a stack and a set \textit{visited}.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \textit{visited}:
    - add \( u \) to \textit{visited}.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \{\( v_1, v_4, v_3, v_6 \)\}
Depth First Search (DFS) with Visited Set

Use a stack and a set \textit{visited}.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \gets \text{stack.pop()} \)
  - if \( u \) is not in \textit{visited}:
    - add \( u \) to \textit{visited}.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \( \{v_1, v_4, v_3, v_6\} \)
Depth First Search (DFS) with \textit{Visited} Set

Use a stack and a set \textit{visited}.

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \textit{visited}:
    - add \( u \) to \textit{visited}.
    - push all vertices adjacent to \( u \) to the stack.

Visited: \( \{v_1, v_4, v_3, v_6, v_7\} \)
Depth First Search (DFS) with Visited Set

Use a stack and a set `visited`.

- Push `s` to the stack.
- While the stack is not empty:
  - `u` <- stack.pop()
  - if `u` is not in `visited`:
    - add `u` to `visited`.
    - push all vertices adjacent to `u` to the stack.

Visited: `{v₁, v₄, v₃, v₆, v₇}`
Depth First Search (DFS) with Visited Set

Use a stack and a set visited.

- Push s to the stack.
- While the stack is not empty:
  - $u \leftarrow$ stack.pop()
  - if $u$ is not in visited:
    - add $u$ to visited.
    - push all vertices adjacent to $u$ to the stack.

Visited: \{v_1, v_4, v_3, v_6, v_7, v_5\}
Depth First Search (DFS) with Visited Set

Use a stack and a set \( \text{visited} \).

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \( \text{visited} \):
    - add \( u \) to \( \text{visited} \).
    - push all vertices adjacent to \( u \) to the stack.

Visited: \( \{v_1,v_4,v_3,v_6,v_7,v_5\} \)
Depth First Search (DFS) with \textit{Visited} Set

Use a stack and a set \textit{visited}.

- Push $s$ to the stack.
- While the stack is not empty:
  - $u \leftarrow$ stack.pop()
  - if $u$ is not in \textit{visited}:
    - add $u$ to \textit{visited}.
    - push all vertices adjacent to $u$ to the stack.

\textbf{Visited:} \{v_1, v_4, v_3, v_6, v_7, v_5, v_2\}
Depth First Search (DFS) with Visited Set

Use a stack and a set \( visited \).

- Push \( s \) to the stack.
- While the stack is not empty:
  - \( u \leftarrow \text{stack.pop()} \)
  - if \( u \) is not in \( visited \):
    - add \( u \) to \( visited \).
    - push all vertices adjacent to \( u \) to the stack.

Running time: \( O(|E|) \)

Visited: \( \{v_1, v_4, v_3, v_6, v_7, v_5, v_2\} \)
Recursive DFS (with \textit{visited} marker kept on vertex objects)

\begin{algorithm}
\SetAlgoLined
void dfs( Vertex v )
\{ \\
\hspace{1em} v.visited = true; \\
\hspace{1em} for each Vertex w adjacent to v \\
\hspace{2em} if( \neg w.visited ) \\
\hspace{3em} dfs( w ); \\
\}
\end{algorithm}

DFS Spanning Tree \\
$V_1$
Recursive DFS (with \textit{visited} marker kept on vertex objects)

```c
void dfs( Vertex v ) {
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

DFS Spanning Tree
Recursive DFS (with \textit{visited} marker kept on vertex objects)

\begin{verbatim}
void dfs( Vertex v ) {
  v.visited = true;
  for each Vertex w adjacent to v
    if( !w.visited )
      dfs( w );
}
\end{verbatim}

DFS Spanning Tree
Recursive DFS (with *visited* marker kept on vertex objects)

```java
void dfs( Vertex v ) {
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

DFS Spanning Tree

```
 DFS Spanning Tree
```

```
 void dfs( Vertex v ) {
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```
Recursive DFS (with \textit{visited} marker kept on vertex objects)

\begin{verbatim}
void dfs( Vertex v ) {
  v.visited = true;
  for each Vertex w adjacent to v
    if( !w.visited )
      dfs( w );
}
\end{verbatim}

DFS Spanning Tree
Recursive DFS (with visited marker kept on vertex objects)

```java
void dfs( Vertex v ) {
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

DFS Spanning Tree
Recursive DFS (with \textit{visited} marker kept on vertex objects)

```java
void dfs( Vertex v ) {
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

DFS Spanning Tree
DFS on the Entire Graph

- It is possible that not all vertices are reachable from a designated start vertex.
DFS on the Entire Graph

• It is possible that not all vertices are reachable from a designated start vertex.
DFS on the Entire Graph

- It is possible that not all vertices are reachable from a designated start vertex.
DFS on the Entire Graph

• It is possible that not all vertices are reachable from a designated start vertex.

V1: V2, V4
V2: V4, V5
V3: V1, V6
V4: V5
V5: V6: V7
V7: V5
DFS on the Entire Graph

• It is possible that not all vertices are reachable from a designated start vertex.

V1: V2, V4
V2: V4, V5
V3: V1, V6
V4: V5
V5: 
V6: V7
V7: V5

• If stack is empty or we reach top of recursion, scan through adjacency list until we find an unseen starting node.
DFS on the Entire Graph

• It is possible that not all vertices are reachable from a designated start vertex.

Running time for complete DFS traversal: $O(|V| + |E|)$

• If stack is empty or we reach top of recursion, scan through adjacency list until we find an unseen starting node.
Breadth-First Search (BFS)

Use a **queue** and a set **visited**.
- Enqueue $s$
- Add $s$ to **visited**
- While the queue is not empty:
  - $u \leftarrow$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v$ is not in **visited**:
      - Add $v$ to visited.
      - enqueue($v$).

Queue $V_1$

Visited: $\{V_1\}$
Breadth-First Search (BFS)

Use a queue and a set visited.
- Enqueue s
- Add s to visited
- While the queue is not empty:
  - u <- dequeue()
  - for each vertex v that is adjacent to u:
    - if v is not in visited:
      - Add v to visited.
      - enqueue(v).

Queue  \[ V_2 \ V_4 \]

Visited: \( \{ v_1, v_2, v_4 \} \)
Breadth-First Search (BFS)

Use a queue and a set visited.
- Enqueue s
- Add s to visited
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in visited:
      - Add \( v \) to visited.
      - enqueue(\( v \)).

Queue  \( \{V_4, V_5\} \)

Visited:  \( \{V_1, V_2, V_4, V_5\} \)
Breadth-First Search (BFS)

Use a queue and a set \( \text{visited} \).
- Enqueue \( s \)
- Add \( s \) to \( \text{visited} \)
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in \( \text{visited} \):
      - Add \( v \) to \( \text{visited} \).
      - enqueue(\( v \)).
Breadth-First Search (BFS)

Use a queue and a set visited.
- Enqueue s
- Add s to visited
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in visited:
      - Add \( v \) to visited.
      - enqueue(\( v \)).
Breadth-First Search (BFS)

Use a queue and a set \textit{visited}.
- Enqueue \( s \)
- Add \( s \) to \textit{visited}
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in \textit{visited}:
      - Add \( v \) to \textit{visited}.
      - enqueue(v).
Breadth-First Search (BFS)

Use a queue and a set \( \text{visited} \).
- Enqueue \( s \)
- Add \( s \) to \( \text{visited} \)
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in \( \text{visited} \):
      - Add \( v \) to \( \text{visited} \).
      - enqueue(\( v \)).

Queue

\[ \text{Queue: } \{ V_3 \} \]

Visited: \( \{ V_1, V_2, V_4, V_5, V_7, V_6, V_3 \} \)
Breadth-First Search (BFS)

Use a queue and a set \textit{visited}.
- Enqueue \( s \)
- Add \( s \) to \textit{visited}
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \) is not in \textit{visited}:
      - Add \( v \) to \textit{visited}.
      - enqueue(\( v \)).

Queue

Visited:\{\( v_1, v_2, v_4, v_5, v_7, v_6, v_3 \}\}
Breadth-First Search (BFS)

Use a queue and a set visited.
- Enqueue $s$
- Add $s$ to visited
- While the queue is not empty:
  - $u \leftarrow$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v$ is not in visited:
      - Add $v$ to visited.
      - enqueue($v$).

Running time (to traverse the entire graph): $O(|V|+|E|)$

Queue

Visited: \{v_1, v_2, v_4, v_5, v_7, v_6, v_3\}
Breadth-First Search (BFS)

Use a queue and a set \textit{visited}.
- Enqueue \( s \)
- Add \( s \) to \textit{visited}
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)

BFS will traverse the entire graph even without a visited set.
DFS can get stuck in a loop.

Visited: \( \{v_1, v_2, v_4, v_5, v_7, v_6, v_3\} \)

Queue

Running time (to traverse the entire graph): \( O(|V|+|E|) \)
Finding Shortest Paths

• Goal: Find the shortest path between two vertices s and t.

What is the shortest path between \(v_3\) and \(v_7\)?
Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.

What is the shortest path between \( v_3 \) and \( v_7 \)?
Finding Shortest Paths

• Goal: Find the shortest path between two vertices s and t.

• It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called **single-source shortest paths**.
Finding Shortest Paths

• Goal: Find the shortest path between two vertices s and t.

• It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called single-source shortest paths.
Finding Shortest Paths

- Goal: Find the shortest path between two vertices \( s \) and \( t \).
- It turns out that finding the shortest path between \( s \) and \( \text{ALL} \) other vertices is just as easy. This problem is called **single-source shortest paths**.
Finding Shortest Paths with BFS

- s.distance = 0
- for all v ∈ V set v.distance = ∞
- enqueue s
- While the queue is not empty:
  - u <- dequeue()
  - for each vertex v that is adjacent to u:
    - if v.distance == ∞
      - v.distance = u.distance + 1
      - enqueue(v)
Finding Shortest Paths with BFS

- \( s.\text{distance} = 0 \)
- For all \( v \in V \) set \( v.\text{distance} = \infty \)
- Enqueue \( s \)
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue}() \)
  - For each vertex \( v \) that is adjacent to \( u \):
    - If \( v.\text{distance} == \infty \)
    - \( v.\text{distance} = u.\text{distance} + 1 \)
    - Enqueue \( v \)
Finding Shortest Paths with BFS

- s.distance = 0
- for all $v \in V$ set $v$.distance = $\infty$
- enqueue s
- While the queue is not empty:
  - $u \leftarrow$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v$.distance == $\infty$
      - $v$.distance = $u$.distance + 1
      - enqueue($v$)
Finding Shortest Paths with BFS

- $s$.distance = 0
- for all $v \in V$ set $v$.distance = $\infty$
- enqueue $s$
- While the queue is not empty:
  - $u$ <- dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v$.distance == $\infty$
      - $v$.distance = $u$.distance + 1
      - enqueue($v$)
Finding Shortest Paths with BFS

- s.distance = 0
- for all \( v \in V \) set \( v.d\text{istance} = \infty \)
- enqueue s
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v.d\text{istance} == \infty \)
    - \( v.d\text{istance} = u.d\text{istance} + 1 \)
    - enqueue(\( v \))

Queue: \[ V_4, V_5 \]
Finding Shortest Paths with BFS

- \( s \text{.distance} = 0 \)
- for all \( v \in V \) set \( v \text{.distance} = \infty \)
- enqueue \( s \)
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v \text{.distance} == \infty \)
    - \( v \text{.distance} = u \text{.distance} + 1 \)
    - enqueue\( (v) \)
Finding Shortest Paths with BFS

- s.distance = 0
- for all v ∈ V set v.distance = ∞
- enqueue s
- While the queue is not empty:
  - u ← dequeue()
  - for each vertex v that is adjacent to u:
    - if v.distance == ∞
      - v.distance = u.distance + 1
      - enqueue(v)
Finding Shortest Paths with BFS

- \( s\.distance = 0 \)
- for all \( v \in V \) set \( v\.distance = \infty \)
- enqueue \( s \)
- While the queue is not empty:
  - \( u \leftarrow \text{dequeue()} \)
  - for each vertex \( v \) that is adjacent to \( u \):
    - if \( v\.distance == \infty \)
    - \( v\.distance = u\.distance + 1 \)
    - enqueue(u)
Finding Shortest Paths with BFS

- $s$.distance = 0
- for all $v \in V$ set $v$.distance = $\infty$
- enqueue $s$
- While the queue is not empty:
  - $u \leftarrow$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v$.distance == $\infty$
      - $v$.distance = $u$.distance + 1

This is just BFS. Running time: $O(|V|+|E|)$
Finding Shortest Paths with BFS - Back pointers

Maintain pointers to the previous node on the shortest path.

- s.distance = 0
- for all $v \in V$ set $v.$distance = $\infty$
- enqueue $s$
- While the queue is not empty:
  - $u \leftarrow$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v.$distance == $\infty$
      - $v.$prev = $u$
      - $v.$distance = $u.$distance + 1
      - enqueue($v$)
Finding Shortest Paths with BFS - Back pointers

Maintain pointers to the previous node on the shortest path.

- $s.distance = 0$
- for all $v \in V$ set $v.distance = \infty$
- enqueue $s$
- While the queue is not empty:
  - $u \gets$ dequeue()
  - for each vertex $v$ that is adjacent to $u$:
    - if $v.distance == \infty$
      - $v.prev = u$
      - $v.distance = u.distance + 1$
      - enqueue($v$)

Queue: $V_6 \ V_2 \ V_4$
Weighted Shortest Paths

- Goal: Find the shortest path between two vertices $s$ and $t$.

What is the shortest path between $v_2$ and $v_6$?
Weighted Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- Normal BFS will find this path.

What is the shortest path between $v_2$ and $v_6$?
Weighted Shortest Paths

- Goal: Find the shortest path between two vertices $s$ and $t$.
- This path is shorter.

What is the shortest path between $v_2$ and $v_6$?

What is the shortest path between $v_2$ and $v_6$?
Negative Weights

• We normally expect the shortest path to be simple.
• Edges with Negative Weights can lead to negative cycles.
• The concept of “shortest path” is then not clearly defined.

What is the shortest path between $v_2$ and $v_6$?
Dijkstra’s Algorithm for Weighted Shortest Path
Dijkstra’s Algorithm for Weighted Shortest Path

• Cost annotations for each vertex reflect the lowest cost using only vertices visited so far.
Dijkstra’s Algorithm for Weighted Shortest Path

• Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*

• That means there might be a lower-cost path through other vertices that have not been seen yet.
Dijkstra’s Algorithm for Weighted Shortest Path

• Cost annotations for each vertex reflect the lowest cost using only vertices visited so far.

• That means there might be a lower-cost path through other vertices that have not been seen yet.

• Keep nodes on a priority queue and always expand the vertex with the lowest cost annotation first!

• Intuitively, this means we will never overestimate the cost and miss lower-cost path.
Dijkstra’s Algorithm for Weighted Shortest Path

• Cost annotations for each vertex reflect the lowest cost using only vertices visited so far.

• That means there might be a lower-cost path through other vertices that have not been seen yet.

• Keep nodes on a priority queue and always expand the vertex with the lowest cost annotation first! ← This is a greedy algorithm

• Intuitively, this means we will never overestimate the cost and miss lower-cost path.
Dijkstra’s Algorithm

Use a **Priority Queue** $q$
- for all $v \in V$
  - set $v.\text{cost} = \infty$, set $v.\text{visited} = \text{false}$
- $s.\text{cost} = 0$, $s.\text{visited} = \text{true}$;
- $q.\text{insert}(s)$
- While $q$ is not empty:
  - $u \leftarrow q.\text{deleteMin}()$
  - $u.\text{visited} = \text{true}$
  - for each edge $(u, v)$:
    - if not $v.\text{visited}$:
      - if $(u.\text{cost} + \text{cost}(u,v) < v.\text{cost})$
        - $v.\text{cost} = u.\text{cost} + \text{cost}(u,v)$
        - $v.\text{prev} = u$
        - $q.\text{insert}(v)$
Dijkstra’s Algorithm

Use a **Priority Queue** $q$

- for all $v \in V$
  - set $v\text{.cost} = \infty$, set $v\text{.visited} = \text{false}$
- $s\text{.cost} = 0$, $s\text{.visited} = \text{true}$;
- $q\text{.insert}(s)$

- While $q$ is not empty:
  - $u \leftarrow q\text{.deleteMin}()$
  - $u\text{.visited} = \text{true}$
  - for each edge $(u,v)$:
    - if not $v\text{.visited}$:
      - if $(u\text{.cost} + \text{cost}(u,v) < v\text{.cost})$
        - $v\text{.cost} = u\text{.cost} + \text{cost}(u,v)$
        - $v\text{.prev} = u$
        - $q\text{.insert}(v)$
Dijkstra’s Algorithm

Use a **Priority Queue** $q$
- for all $v \in V$
  - set $v.\text{cost} = \infty$, set $v.\text{visited} = \text{false}$
- $s.\text{cost} = 0$, $s.\text{visited} = \text{true}$;
- $q.\text{insert}(s)$

- While $q$ is not empty:
  - $u \leftarrow q.\text{deleteMin}()$
  - $u.\text{visited} = \text{true}$
  - for each edge $(u,v)$:
    - if not $v.\text{visited}$:
      - if $(u.\text{cost} + \text{cost}(u,v) < v.\text{cost})$
        - $v.\text{cost} = u.\text{cost} + \text{cost}(u,v)$
        - $v.\text{prev} = u$
        - $q.\text{insert}(v)$
Dijkstra’s Algorithm

Use a Priority Queue $q$

- for all $v \in V$ set $v.cost = \infty$, set $v.visited = false$
- $s.cost = 0$, $s.visited = true$
- $q.insert(s)$

- While $q$ is not empty:
  - $u <- q.deleteMin()$
  - $u.visited = true$
  - for each edge $(u,v)$:
    - if not $v.visited$:
      - if $(u.cost + cost(u,v) < v.cost)$
        - $v.cost = u.cost + cost(u,v)$
        - $v.prev = u$
        - $q.insert(v)$
Dijkstra’s Algorithm

Use a **Priority Queue** $q$

- for all $v \in V$
  
  set $v.cost = \infty$, set $v.visited = false$

- $s.cost = 0$, $s.visited = true$

- $q.insert(s)$

- **While** $q$ is not empty:
  
  - $u <- q.deleteMin()$

  - $u.visited = true$

  - for each edge $(u, v)$:

    - if not $v.visited$:
      
      - if $(u.cost + \text{cost}(u,v) < v.cost)$

      - $v.cost = u.cost + \text{cost}(u,v)$

      - $v.prev = u$

    - $q.insert(v)$
Dijkstra’s Algorithm

Use a **Priority Queue** \( q \)

- for all \( v \in V \)
  - set \( v.cost = \infty \), set \( v.visited = \text{false} \)
- \( s.cost = 0 \), \( s.visited = \text{true} \);
- \( q.insert(s) \)

- While \( q \) is not empty:
  - \( u \leftarrow q.deleteMin() \)
  - \( u.visited = \text{true} \)
  - for each edge \((u,v)\):
    - if not \( v.visited \):
      - if \((u.cost + \text{cost}(u,v) < v.cost)\)
        - \( v.cost = u.cost + \text{cost}(u,v) \)
        - \( v.prev = u \)
        - \( q.insert(v) \)
Dijkstra’s Algorithm

Use a **Priority Queue** $q$

- for all $v \in V$
  - set $v\.cost = \infty$, set $v\.visited = false$
- $s\.cost = 0$, $s\.visited = true$
- $q\.insert(s)$

- While $q$ is not empty:
  - $u <- q\.deleteMin()$
  - $u\.visited = true$
  - for each edge $(u, v)$:
    - if not $v\.visited$:
      - if $(u\.cost + cost(u, v) < v\.cost)$
        - $v\.cost = u\.cost + cost(u, v)$
        - $v.prev = u$
        - $q\.insert(v)$
Dijkstra’s Algorithm

Use a **Priority Queue** $q$

- for all $v \in V$
  - set $v\.cost = \infty$, set $v\.visited = \text{false}$
- $s\.cost = 0$, $s\.visited = \text{true}$;
- $q\.insert(s)$

- While $q$ is not empty:
  - $u <- q\.deleteMin()$
  - $u\.visited = \text{true}$
  - for each edge $(u, v)$:
    - if not $v\.visited$:
      - if $(u\.cost + \text{cost}(u, v) < v\.cost)$
        - $v\.cost = u\.cost + \text{cost}(u, v)$
        - $v\.prev = u$
        - $q\.insert(v)$
• While $q$ is not empty:
  • $u \leftarrow q$.deleteMin()
  • $u$.visited = true
  • for each edge $(u, v)$:
    • if not $v$.visited:
      • if $(u$.cost $+ \text{cost}(u, v) < v$.cost)
        • $v$.cost = $u$.cost $+ \text{cost}(u, v)$
        • $v$.prev = $u$
        • $q$.insert($v$)

Dijkstra’s Algorithm - a subtle bug

$v_7$.cost $+ \text{cost}(v_6, v_7) = 6$
• While $q$ is not empty:
  • $u \leftarrow q$.deleteMin()
  • $u$.visited = true
  • for each edge $(u,v)$:
    • if not $v$.visited:
      • if ($u$.cost + cost($u,v$) < $v$.cost)
        • $v$.cost = $u$.cost + cost($u,v$)
        • $v$.prev = $u$
        • $q$.insert($v$)

$v_7$.cost + cost($v_6,v_7$) = 6

• $v_7$ is already in $q$, and has not been visited.
• does insert($v_7$) create a new entry in the $q$ or update the existing one?
• if $q$ is a heap, updating the cost will change $v_7$ everywhere in the heap and might make the heap invalid.
Dijkstra’s Algorithm - Fixed

Use a **Priority Queue** \( q \):
- for all \( v \in V \)
  - set \( v.\text{cost} = \infty \), set \( v.\text{visited} = \text{false} \)
- \( s.\text{cost} = 0 \)
- \( q.\text{insert}((0, s)) \)

While \( q \) is not empty:
- \((\text{cost}_u, u) <- q.\text{deleteMin}()\)
  - **if not** \( u.\text{visited} \):
    - \( u.\text{visited} = \text{true} \)
    - for each edge \((u, v)\):
      - **if not** \( v.\text{visited} \):
        - **if** \((\text{cost}_u + \text{cost}(u, v) < v.\text{cost})\)
          - \( v.\text{cost} = u.\text{cost} + \text{cost}(u, v) \)
          - \( v.\text{prev} = u \)
          - \( q.\text{insert}((v.\text{cost}, v)) \)

- Keep a separate cost object in the queue that isn’t updated.
- Ignore duplicate entries for vertices.

\[ v_7.\text{cost} + \text{cost}(v_6, v_7) = 6 \]
Dijkstra’s Running Time

• There are $|E|$ insert and deleteMin operations.

• The maximum size of the priority queue is $O(|E|)$. Each insert takes $O(\log |E|)$

$O(|E| \log |E|)$

Use a Priority Queue $q$

• for all $v \in V$
  - set $v.cost = \infty$, set $v.visited = false$
• $s.cost = 0$
• $q.insert((0, s))$

• While $q$ is not empty:
  • $(costu, u) \leftarrow q.deleteMin()$
  • if not $u.visited$:
    • $u.visited = true$
    • for each edge $(u,v)$:
      • if not $v.visited$:
        • if $(ucost + cost(u,v) < v.cost)$
          • $v.cost = u.cost + cost(u,v)$
          • $v.prev = u$
          • $q.insert((v.cost, v))$
Dijkstra’s Running Time

- There are $|E|$ insert and deleteMin operations.
- The maximum size of the priority queue is $O(|E|)$. Each insert takes $O(\log |E|)$

Use a Priority Queue $q$

- for all $v \in V$
  set $v$.cost = $\infty$, set $v$.visited = false
- s.cost = 0
- $q$.insert((0, s))

- While $q$ is not empty:
  - $(\text{cost}_u, u) \leftarrow q$.deleteMin()
  - if not $u$.visited:
    - $u$.visited = true
    - for each edge $(u, v)$:
      - if not $v$.visited:
        - if $(\text{ucost} + \text{cost}(u, v) < v$.cost)
          - $v$.cost = $u$.cost + $\text{cost}(u, v)$
          - $v$.prev = $u$
          - $q$.insert(($v$.cost, v))

because $|E| \leq |V|^2$, and therefore $\log |E| \leq 2 \log |V|$
A topological sort of a DAG is an ordering of its vertices such that if there is a path from $u$ to $w$, $u$ appears before $w$ in the ordering.

Topological Sort in DAGs
Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from $u$ to $w$, $u$ appears before $w$ in the ordering.
Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from $u$ to $w$, $u$ appears before $w$ in the ordering.

W1007  W1004  W3134  W3203  W3137  W3157
Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from $u$ to $w$, $u$ appears before $w$ in the ordering.
Topological Sort in DAGs

A topological sort of a DAG is an ordering of its vertices such that if there is a path from \( u \) to \( w \), \( u \) appears before \( w \) in the ordering.

W1007  W1004  W3134  W3203  W3137  W3157  W3261  W4111  W4701  W4115  W4156
Application: Critical Path Analysis

• An **Event-Node Graph** is a DAG in which
• Edges represent tasks, weight represents the time it takes to complete the task.
• Vertices represent the event of completing a set of tasks.
Application: Critical Path Analysis

- We are interested in the earliest completion time. (Earliest time we can reach the final event).
- This is equivalent to finding the *longest* path through the DAG (why does this not work with cycles?).
Application: Critical Path Analysis

- If an event has more than one incoming event, all tasks have to be finished before other tasks can proceed.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

- Basic idea: Compute the earliest completion time for each event.
- Can use Dijkstra’s algorithm $O(|E| \log |V|)$.
- We now try to find the longest path.
Application: Critical Path Analysis

- Basic idea: Compute the earliest completion time for each event.
- Can use Dijkstra’s algorithm $O(|E| \log |V|)$.
- We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

- Basic idea: Compute the earliest completion time for each event.
- Can use Dijkstra’s algorithm $O(|E| \log |V|)$.
- We now try to find the longest path.
Application: Critical Path Analysis

• Basic idea: Compute the earliest completion time for each event.

• Can use Dijkstra’s algorithm $O(|E| \log |V|)$.

• We now try to find the longest path.
Application: Critical Path Analysis

- For DAGs we can improve on Dijkstra’s $O(|E| \log |V|)$ bound.
- Use topological sort.
Application: Critical Path Analysis

- Basic idea: Compute the earliest completion time for each event.

- Process events in topological order.

Need at least 5 time steps to get here
Computing Topological Order

- Basic idea: Use BFS!
- To compute topological order, we need to find all incoming edges to a node first before visiting the node.
Computing Topological Order

- Example Application: Computing earliest completion time.

- First annotate each vertex with the number of incoming edges (the **indegree**).
Computing Topological Order

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v1

Output:
Computing Topological Order

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v2 v3
Output: v1
Computing Topological Order

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v3 v4

Output: v1 v2
Computing Topological Order

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v4
Output: v1 v2 v3
Computing Topological Order

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v5
Output: v1 v2 v3 v4
Computing Topological Order

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v6 v9
Output: v1 v2 v3 v4 v5
Computing Topological Order

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v9 v7

Output: v1 v2 v3 v4 v5 v6
While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.

If the indegree of any new vertex becomes 0, enqueue it.

Queue: v7

Output: v1 v2 v3 v4 v5 v6 v9
Computing Topological Order

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v8

Output: v1 v2 v3 v4 v5 v6 v9 v7
While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.

If the indegree of any new vertex becomes 0, enqueue it.

Queue:

Output: v1, v2, v3, v4, v5, v6, v9, v7, v8
Topological Sort - Running Time

- First annotate each vertex with its **indegree**.
- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.
Topological Sort - Running Time

- First annotate each vertex with its **indegree**.
- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes.
- If the indegree of any new vertex becomes 0, enqueue it.

This is just BFS. Running time: $O(|V|+|E|)$
Earliest Completion Time

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**

• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v1

Output:
Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v2 v3

Output: v1
While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**

If the indegree of any new vertex becomes 0, enqueue it.

Queue: v3 v4

Output: v1 v2
Early Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v4

Output: v1 v2 v3
Earliest Completion Time

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**

• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v5

Output: v1 v2 v3 v4
Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**

- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v6 v9

Output: v1 v2 v3 v4 v5
Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v9 v7

Output: v1 v2 v3 v4 v5 v6
Earliest Completion Time

• While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**

• If the indegree of any new vertex becomes 0, enqueue it.

Queue: v7

Output: v1 v2 v3 v4 v5 v6 v9
Earliest Completion Time

- While the queue is not empty, dequeue a vertex, print it and decrement the indegree of its adjacent nodes. **Update earliest completion time for each adjacent node.**
- If the indegree of any new vertex becomes 0, enqueue it.

Queue: v8

Output: v1 v2 v3 v4 v5 v6 v9 v7