Chapter 20. Elementary Graph Algorithms & 22. Shortest Paths

Joon Soo Yoo

June 5, 2025

Assignment

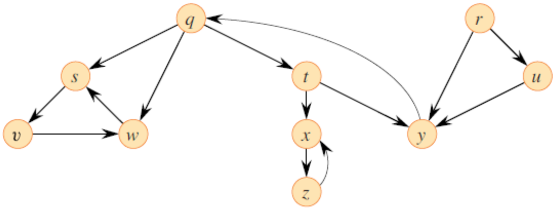
- Read §20.3, 22.0
- Problems
  - §20.3 - 5.(a), (b)

Chapter 20: Elementary Graph Algorithms

- Chapter 20.1: Representations of Graphs
- Chapter 20.2: Breadth-First Search
- **Chapter 20.3: Depth-First Search**
- Chapter 20.4: Topological Sort
- Chapter 20.5: Strongly Connected Components

What is DFS?

- DFS explores as deep as possible before backtracking.
- Recursively visits unexplored neighbors.
- Restarts from a new source if unvisited vertices remain.
- Produces a **DFS forest** instead of a single tree.



DFS Forest and Predecessor Subgraph

- Each vertex v has a predecessor $v.\pi$.
- The predecessor subgraph is:

$$G_\pi = (V, E_\pi) \quad \text{where} \quad E_\pi = \{(v.\pi, v) \mid v \in V,\ v.\pi \neq \text{NIL}\}$$

- G_π forms a **depth-first forest**.

Vertex Coloring in DFS

- **WHITE**: Not yet visited
- **GRAY**: Discovered, currently exploring
- **BLACK**: Fully explored
- Ensures that each vertex belongs to exactly one DFS tree

Timestamps in DFS

- Each vertex v is assigned two timestamps:
  - $v.d$ (discovery time)
  - $v.f$ (finish time)
- Timestamps help in:
  - Classifying edge types (tree, back, forward, cross)
  - Detecting cycles
  - Topological sorting

DFS Timestamps

- DFS assigns each vertex u two timestamps:
  - $u.d$: discovery time (when u is first visited)
  - $u.f$: finish time (after all of u's neighbors are explored)
- Timestamps are integers in $[1, 2|V|]$
- Always: $u.d < u.f$

Vertex State Over Time

- Each vertex u changes state during DFS:
  - **WHITE** before $u.d$
  - **GRAY** between $u.d$ and $u.f$
  - **BLACK** after $u.f$
- These states ensure that vertices are visited and completed correctly

DFS Algorithm

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

DFS Visit Algorithm

```
DFS-VISIT(G, u)
time = time + 1                      // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each vertex v in G.Adj[u]        // explore each edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
time = time + 1
u.f = time
u.color = BLACK                      // blacken u; it is finished
```

Why DFS Order Matters

- DFS depends on:
  - The order of vertices in $G.V$
  - The order of neighbors in each adjacency list $\text{Adj}[u]$
- This affects:
  - The DFS forest structure
  - Timestamps (d, f)
  - Edge classifications (tree, back, forward, cross)
- But all are valid DFS results

DFS-VISIT Calls and Edge Scanning

- `DFS-VISIT(G, u)` is called exactly once per vertex:

$$\text{Total calls to } \texttt{DFS-VISIT} = |V|$$

- Each call explores neighbors:

$$\texttt{for each } v \in \mathrm{Adj}[u]$$

- Total adjacency scans:

$$\sum_{u \in V} |\mathrm{Adj}[u]|$$

  - $= |E|$ for directed graphs
  - $= 2|E|$ for undirected graphs

Final Running Time of DFS

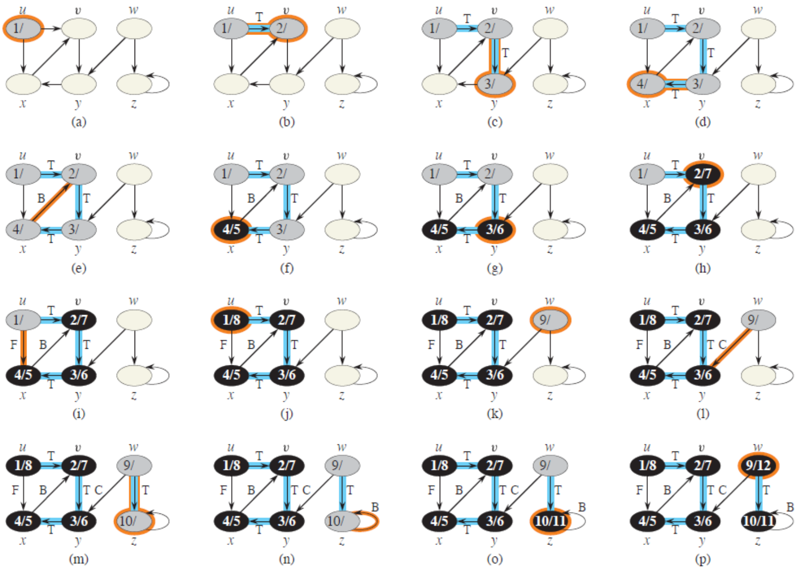
- Initialization: $\Theta(|V|)$
- DFS-VISIT calls: $\Theta(|V|)$
- Neighbor scans: $\Theta(|E|)$

$$\boxed{\Theta(|V| + |E|)}$$

Classifying Edges in DFS

- DFS reveals graph structure by classifying edges:
- **Tree edge**: (u, v) discovers v for the first time
- **Back edge**: (u, v) goes to an ancestor of u (or a self-loop)
- **Forward edge**: (u, v) goes to a proper descendant of u
- **Cross edge**: all other edges (between DFS trees or unrelated nodes)

DFS Diagram



Classification by Color During DFS

When DFS explores edge (u, v):

- v is **WHITE** $\Rightarrow$ Tree edge
- v is **GRAY** $\Rightarrow$ Back edge (ancestor)
- v is **BLACK**:
  - If $u.d < v.d$ and $v.f < u.f$ $\Rightarrow$ Forward edge
  - If $v.f < u.d$ $\Rightarrow$ Cross edge

Why Edge Classification Matters

- Back edges indicate **cycles**
- Forward and cross edges only occur in **directed** graphs
- DFS has enough info to classify edges:
  - via vertex colors at traversal time
  - via discovery/finish times (d, f)
- Helps in graph analysis

Chapter 22: Single-Source Shortest Paths

- Chapter 22.1: The Bellman-Ford Algorithm
- Chapter 22.2: Single-Source Shortest Paths in Directed Acyclic Graphs
- Chapter 22.3: Dijkstra's Algorithm
- Chapter 22.4: Difference Constraints and Shortest Paths
- Chapter 22.5: Proofs of Shortest-Paths Properties

Finding Shortest Routes – Real-World Motivation

- Drive from **NY** to **CA**
- GPS models:
  - Intersections $\rightarrow$ vertices
  - Roads $\rightarrow$ directed edges
  - Distances $\rightarrow$ edge weights
- Enumerating all routes is infeasible

Shortest-Path Problem Setup

Input:

- Directed graph $G = (V, E)$
- Weight function $w : E \to \mathbb{R}$

Path weight:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Shortest-path weight:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{if path exists} \\ \infty & \text{otherwise} \end{cases}$$

Edge Weights – More Than Just Distance

- Weights can represent:
  - Travel time
  - Monetary cost
  - Penalty or loss
  - Any additive metric
- Goal: Minimize total cost along the path

Optimal Substructure of Shortest Paths

- Many shortest-path algorithms rely on the idea of **optimal substructure**.
- That is, a shortest path between two vertices contains other shortest paths within it.
- This property enables the use of:
  - **Greedy algorithms**, e.g., **Dijkstra's algorithm** (Section 22.3)
  - **Dynamic programming**, e.g., **Floyd-Warshall algorithm** (Section 23.2)
- Similar principle is also used in the Edmonds-Karp algorithm (Chapter 24, max flow).

Lemma 22.1 formalizes this optimal-substructure property.

Lemma 22.1: Optimal Substructure of Shortest Paths

Lemma

Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let

$$p = \langle v_0, v_1, \ldots, v_k \rangle$$

be a shortest path from v_0 to v_k. Then for any $0 \leq i \leq j \leq k$, the subpath

$$p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$$

is also a shortest path from v_i to v_j.

Proof of Lemma 22.1

Proof Sketch:

- Suppose p_{ij} is not a shortest path.
- Then there exists another path p'_{ij} with lower weight: $w(p'_{ij}) < w(p_{ij})$.
- Replacing p_{ij} with p'_{ij} in p gives a path from v_0 to v_k with smaller total weight.
- This contradicts the assumption that p is a shortest path.

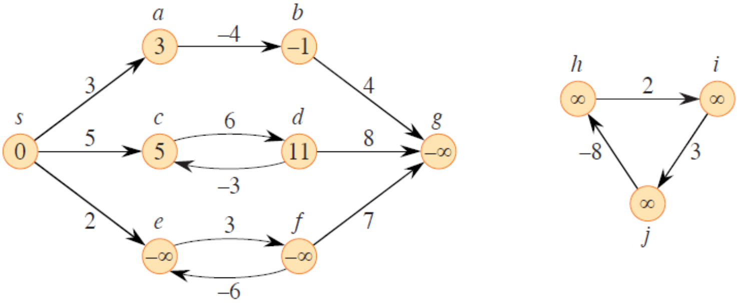
Negative-Weight Edges in Shortest Paths

- Some graphs contain edges with negative weights.
- These do not always cause problems:
  - If there are **no negative-weight cycles reachable from the source** s,

    $$\delta(s, v) \text{ is well-defined for all } v \in V$$

    even if $\delta(s, v) < 0$.
- But if a **negative-weight cycle is reachable from** s:
  - You can cycle repeatedly and lower path weight indefinitely.
  - So: $\delta(s, v) = -\infty$ for any v reachable via that cycle.

Negative-Weight Edges Example



Example: Negative Cycles in Action (Fig. 22.1)

- Example paths:
  - $s \to a$: weight $= 3$, so $\delta(s, a) = 3$
  - $s \to a \to b$: $3 + (-4) = -1$, so $\delta(s, b) = -1$
  - $s \to c$: directly with weight $= 5$
  - $c \to d \to c$: forms a **positive-weight cycle**, so no problem.
- Cycle $e \to f \to e$: weight $= 3 + (-6) = -3 \to$ **negative-weight cycle reachable from** s
- Now we can:
  - Loop arbitrarily through $e \leftrightarrow f$ to decrease cost
  - Then exit to g: $s \to e \to f \to g$
- So: $\delta(s, e) = \delta(s, f) = \delta(s, g) = -\infty$

Impact on Algorithms

- **Dijkstra's algorithm:**
  - Assumes all edge weights ≥ 0
  - Will fail or produce incorrect results with negative-weight edges
- **Bellman-Ford algorithm:**
  - Handles negative weights
  - Produces correct results as long as no negative-weight cycle is reachable from s
  - Can **detect negative-weight cycles**

Summary: Negative-weight edges are okay, but negative-weight cycles are dangerous.

Cycles in Shortest Paths

- Can a shortest path contain a cycle?
- **Negative-weight cycle:** Repeating it reduces the total weight $\rightarrow$ no well-defined shortest path.
- **Positive-weight cycle:** Removing the cycle makes the path shorter:

  $$w(p') = w(p) - w(c) < w(p)$$

  So p wasn't the shortest path.
- **Conclusion:** We can always assume that a shortest path is a **simple path** (i.e., no cycles).

Representing Shortest Paths

Why store paths, not just distances?

- Knowing only the shortest-path *distance* is often not enough.
- Applications (e.g., GPS) require the actual *path*.

Storing paths with `predecessors`**:**

- For each vertex v, store $v.\pi$ (predecessor of v on shortest path).
- Use `PRINT-PATH(G, s, v)` to reconstruct the path from s to v.

Predecessor subgraph:

$$V_\pi = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\}, \quad E_\pi = \{(v.\pi, v) \in E \mid v \in V_\pi \setminus \{s\}\}$$

Shortest-Paths Tree

Definition: A *shortest-paths tree* from source s is a directed subgraph $G' = (V', E')$ such that:

1. V' is the set of vertices reachable from s
2. G' is a rooted tree with root s
3. For all $v \in V'$, the unique path from s to v in G' is a shortest path in G

Key Properties:

- Edge weights are used (unlike BFS trees which use hop-count).
- Shortest paths (and trees) may not be unique.

Shortest-Paths Tree Example

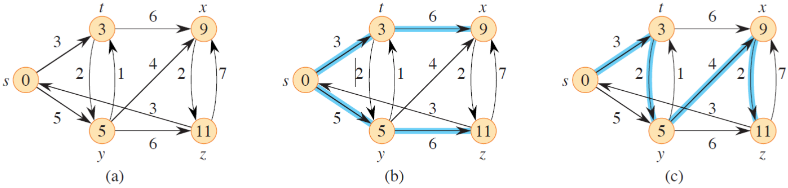


Figure 22.2 (**a**) A weighted, directed graph with shortest-path weights from source s. (**b**) The blue edges form a shortest-paths tree rooted at the source s. (**c**) Another shortest-paths tree with the same root.

Relaxation in Shortest-Paths Algorithms

Core Concept: Relaxation

- Each vertex v maintains a **shortest-path estimate** $v.d$
  - $v.d$ is an upper bound on the weight of the shortest path from source s to v
- The goal is to iteratively **reduce** $v.d$ to the correct shortest-path weight $\delta(s, v)$

Initialization:

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

The RELAX Operation

Relaxing an edge (u, v) with weight $w(u, v)$:

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

Effect:

- Updates $v.d$ if a better path through u is found
- Updates $v.\pi$ to point to u

The RELAX Example

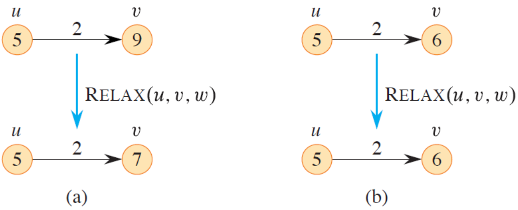


Figure 22.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \le u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

Question?