CSD2183: Data Structures

Single-Source Shortest Paths

Bingjie Xu

Contact: bingjie.xu@singaporetech.edu.sg

9 March 2026

Information

- Homework 1 results will be released by the end of Week 10 (this week).
- Homework 2 has been posted, due on **4 April at 11:59PM** (Week 13). Please refer to the instructions for Homework Project 2 in the xSITe Dropbox.
- The final exam will cover topics from Week 8 - 12, including both lectures and labs.

Overview

	Basics (Week 1 - 6)	Advanced (Week 8 - 13)
1	Foundations	Graph Foundations and Traversal
2	Running Times	Disjoint Sets and Minimum Spanning Trees
3	Sorting	**Shortest Paths (Single-Source)**
4	List and Hash Tables	Dynamic Programming
5	Trees	Greedy Algorithms
6	Consultation	Consultation

Cormen, Thomas H., et al. Introduction to algorithms 4th edition. MIT press, 2022.

Learning Objectives

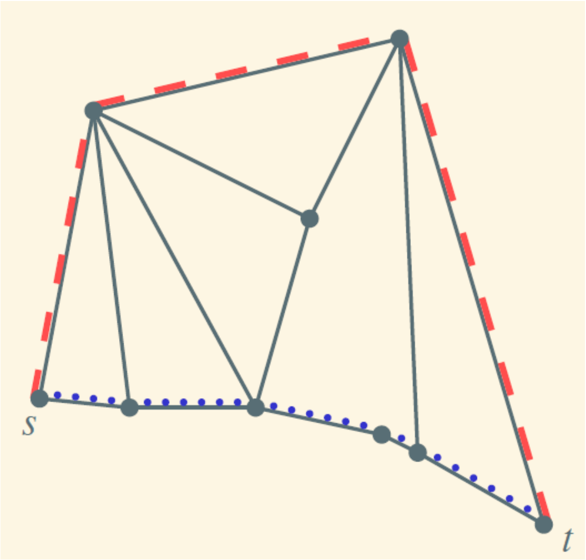
- Define the concept of a **shortest path in a weighted graph**.
- Apply **Dijkstra's and Bellman-Ford** algorithms to a given weighted graph to find a shortest path.
- Argue that Dijkstra's algorithm is correct when none of the edge weights is negative, while Bellman-Ford is general.
- State the growth of the running time as a function of the input size.

Shortest Paths Problem

- Problem formulation
- Use cases

Shortest Paths Problem

- Previously, we used BFS to determine the minimum number of edges connecting two vertices in a graph.
- However, the distance in a graph is not always meaningfully measured by the number of edges. Alternative measures include geometric distance, travel time and monetary cost, represented by edge weights.
- Generalization of BFS to weighted graphs. e.g., MRT route with the minimum time.



- - - path with the fewest edges
- - - path of the least geometric distance

Shortest Paths Problem

How to find the shortest route between two points on a map.

Input:

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

Weight of path $\mathrm{w}(p) = < v_0, v_1, \dots, v_k >$

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

= sum of edge weights on path p.

Shortest-path weight $u \; to \; v$:

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

Shortest path $u \; to \; v$ is any path p such that $w(p) = \delta(u, v)$.

Single-Source Shortest Paths (SSSP)

- In this lecture, we will focus on solving the single-source shortest-paths problem.
- Given a graph $G = (V, E)$, the objective is to find a shortest path from a given source vertex $s \in V$ to every vertex $\mathrm{v} \in V$.

Single-Source Shortest Paths (SSSP)

For each vertex $v \in V$, two key attributes:

- $v.d = \delta(s, v)$.
  - $v.d$ is a *shortest-path estimate* from source s to v, an upper bound on the distance.
  - Initially, $v.d = \infty$.
  - Reduces as algorithms progress. But always maintain $v.d \geq \delta(s, v)$.
- $v.\pi =$ predecessor (parent) of v on a shortest path from s to v.
  - If no predecessor, $v.\pi =$ NIL.
  - π induces a tree – shortest-path tree.

We've seen similar for Prim's. Difference: Prim's $v.d$ indicates the min weights to all the vertices in the MST, here $v.d$ indicates the min weights from s.

SSSP Key Operations

Initialization

$INIT - SINGLE - SOURCE(G, s)$

$for\ each\ v\ \in G.V$

$v.d = \infty$ // shortest path estimate from source

$v.\pi = NIL$ // parent/predecessor vertex

$s.d = 0$

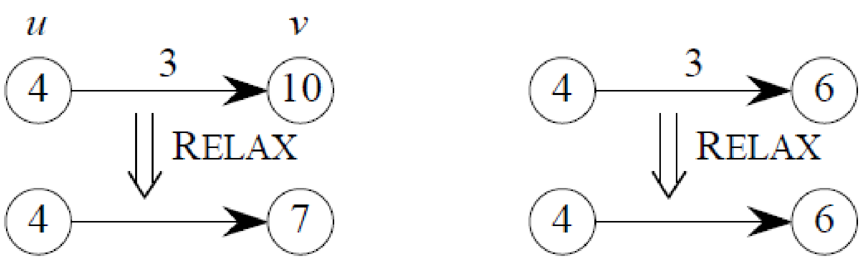
Relaxation

$RELAX(u, v, w)$

$if\ v.d > u.d + w(u, v)$

$v.d = u.d + w(u, v)$

$v.\pi = u$



The algorithms differ in the order and how many times they relax each edge.

Exercise

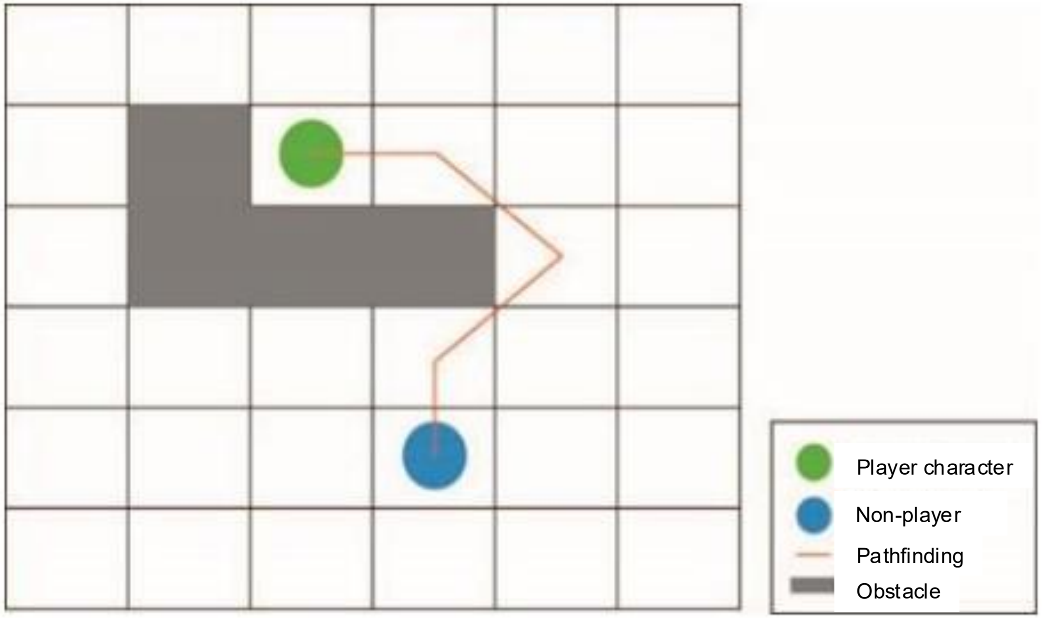
Match each description to the **graph algorithms** that can solve it.

Description
a. Decide reachability from a source vertex to another vertex
b. Connect a source vertex to another vertex with the min cost
c. Connect all vertices with the min cost

Graph Algorithms
1. Graph Traversal
2. Minimum Spanning Trees
3. Shortest Paths (Single-Source)

Graph Algorithm	Description	Example	Graph Construction
Graph Traversal	Reachability from a source vertex to another vertex	Reachability from a source vertex to another vertex. You want to travel from **Bencoolen to Punggol Coast** using the MRT map. Determine whether the destination is reachable.	Both directed or undirected, unweighted
Minimum Spanning Trees	Connect all vertices with the min cost	In city planning, the government designs the **least-cost road network** that connects all major areas of the city.	Connected & undirected graph, weighted
Shortest Paths (Single-Source)	Connect a source vertex to another vertex with the min cost	You want to travel from **Bencoolen to Punggol Coast** using the MRT map. Determine whether the destination is reachable, and if so, find the **minimum travel time and the corresponding route**.	Directed (undirected treated as bidirectional), weighted

Use Case in Game Development



Pathfinding in a video game, such as real-time strategy games, role-playing games, racing games and turn-based strategy games.

In the Non-Player (NPC) context, pathfinding is used to guide between two node points in order to capture the player character.

Pathfinding Algorithms in Game Development

Use Case in Game Development (Lab)

Consider a 3×4 game grid with terrain costs:

```
  0  1  2  3
0 1  5  2  1
1 3  1  4  2
2 1  2  1  3
```

Starting at $(0, 0)$ with cost 1, and target at $(2, 3)$ with cost 3.

Example path: $(0, 0) \rightarrow (0, 1) \rightarrow (1, 1) \rightarrow (2, 1) \rightarrow (2, 2) \rightarrow (2, 3)$

Path cost: $1 + 5 + 1 + 2 + 1 + 3 = 13$ (includes start cell cost)

Note: The optimal path might be different! Use Dijkstra's or Bellman-Ford to find it.

The real-world problem may not be presented in an explicit graph format!

Single-Source Shortest Paths Algorithms

- Dijkstra’s algorithm

Dijkstra's Algorithm

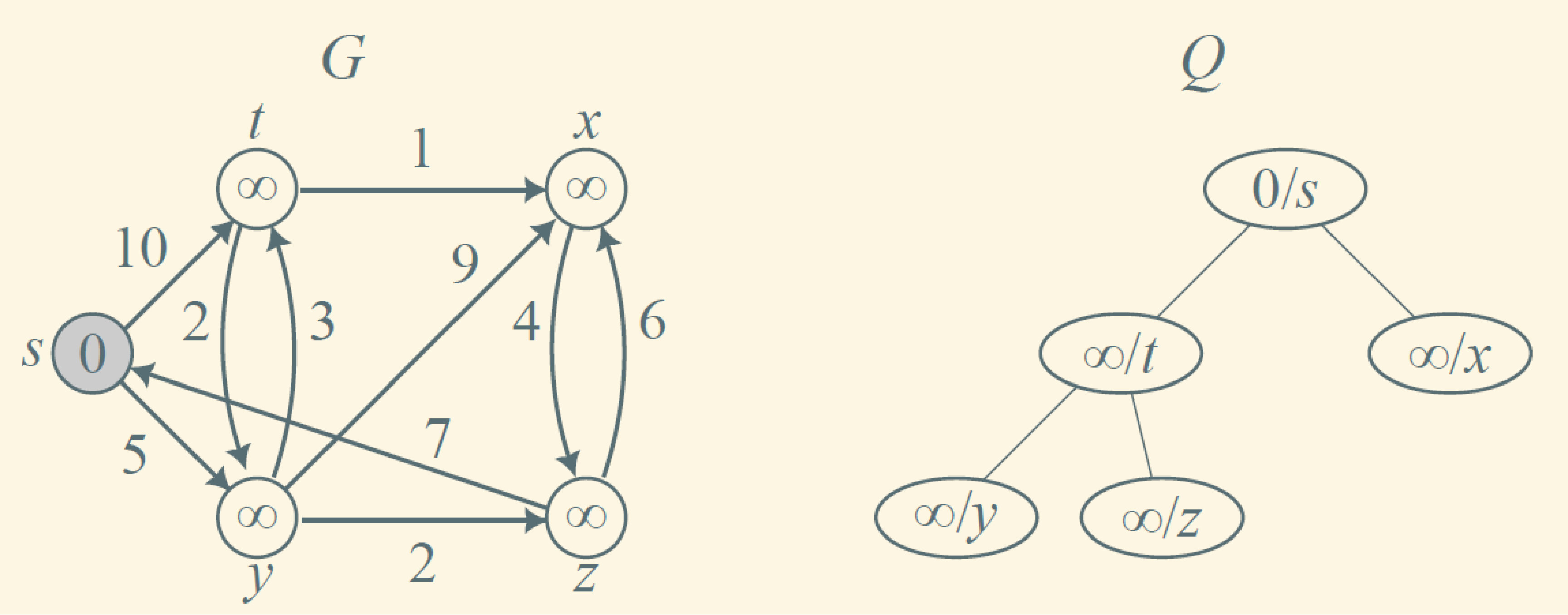
Dijkstra's algorithm finds the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights.

HOW IT WORKS:

1. Initialize: Set distance to source = 0, all others = ∞.
2. Priority Queue: Use a min-heap to always process the vertex with minimum distance.
3. Relax Edges: For each neighbor, if $dist[u] + weight < dist[v]$, update $dist[v]$.
4. Mark Visited: Once processed, mark vertex as visited.
5. Repeat: Continue until priority queue is empty.

Dijkstra's Algorithm: Step-by-Step Illustration

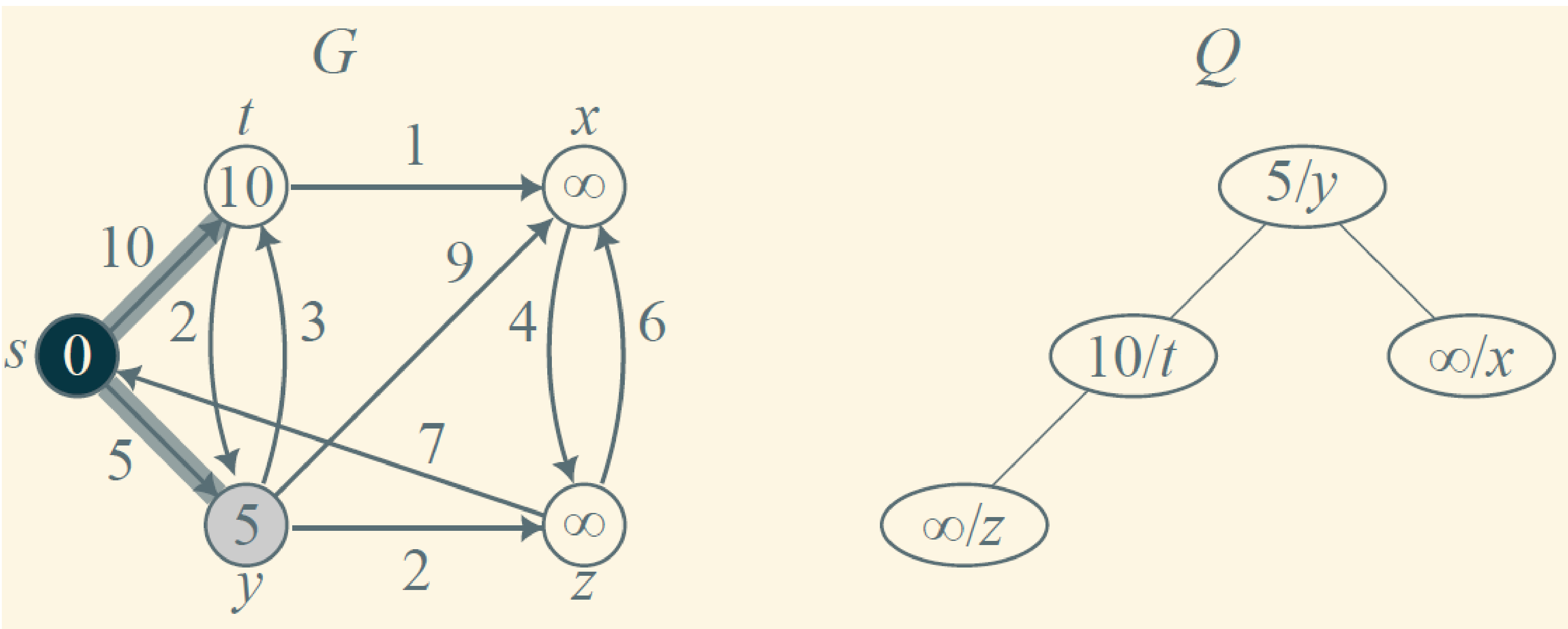
Illustration: Once vertex u with shortest distance is selected, relax for u's connected vertices, greedy method.



The min priority queue structure is initialized arbitrarily except the source s as root.

Dijkstra's Algorithm: Step-by-Step Illustration

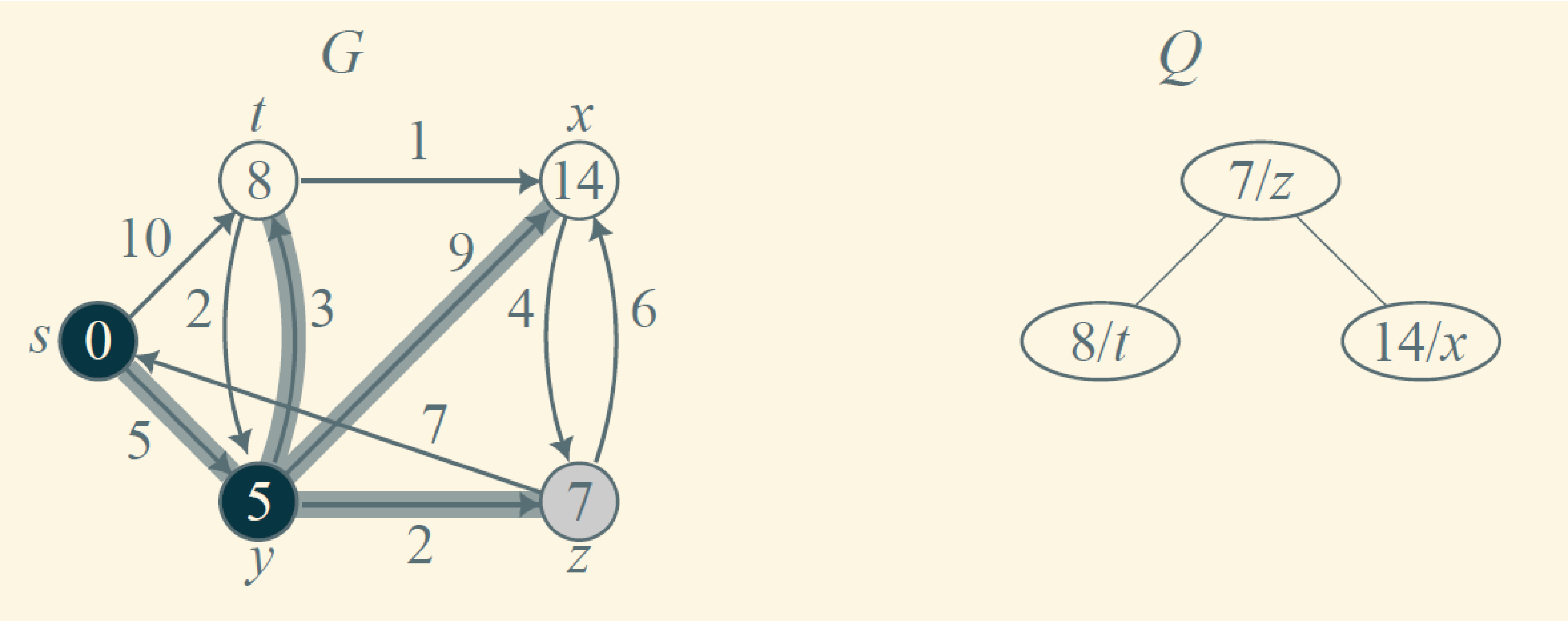
Illustration: Once vertex u with shortest distance is selected, relax for u's connected vertices, greedy method.



Relax edges from s: s → t (weight 10), s → y (weight 5).
Update distances: t = 10, y = 5 (better than ∞).

Dijkstra's Algorithm: Step-by-Step Illustration

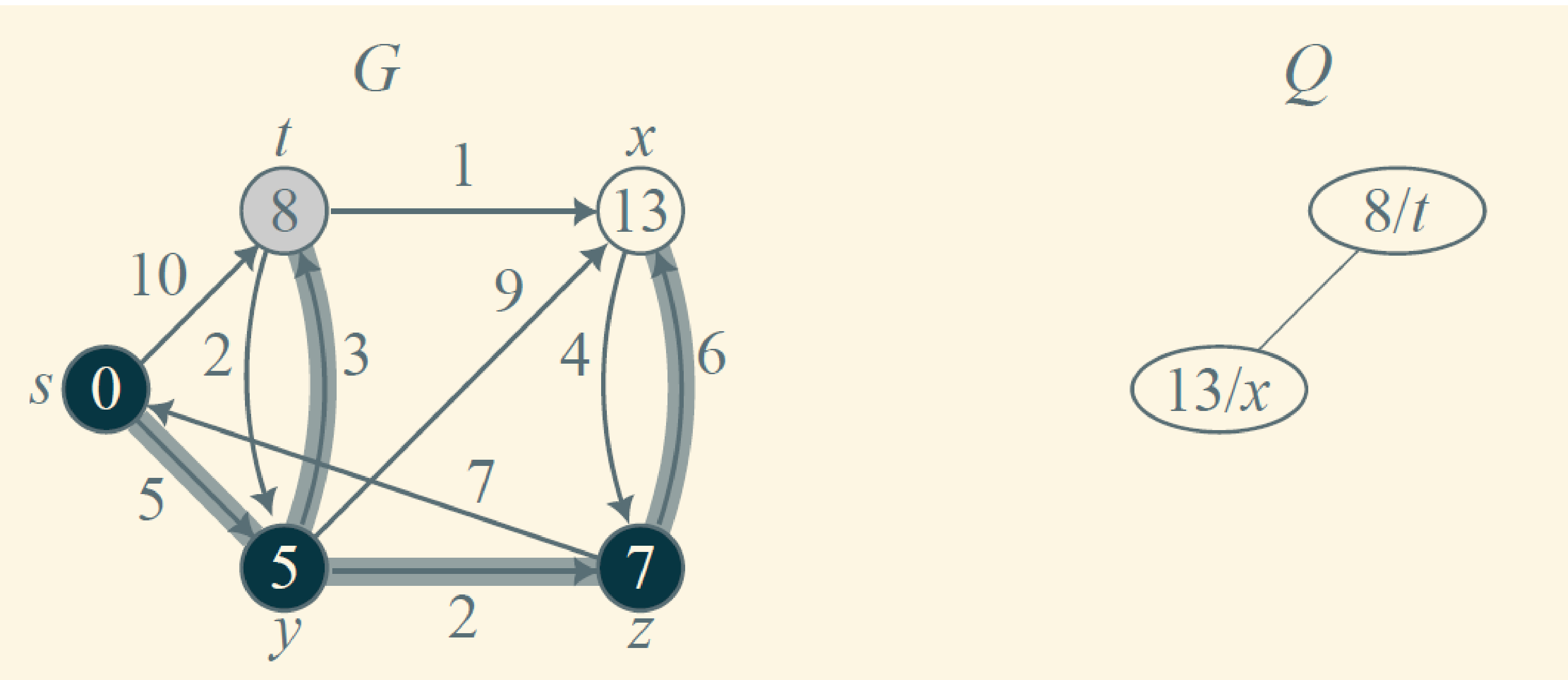
Illustration: Once vertex u with shortest distance is selected, relax for u's connected vertices, greedy method.



Relax edges from y: y → s (weight 5), y → t (3), y → x (9), y → z (2).
Update distances: s done, t = 8 (less than 10), x = 5 + 9 = 14, z = 5 + 2 = 7.

Dijkstra's Algorithm: Step-by-Step Illustration

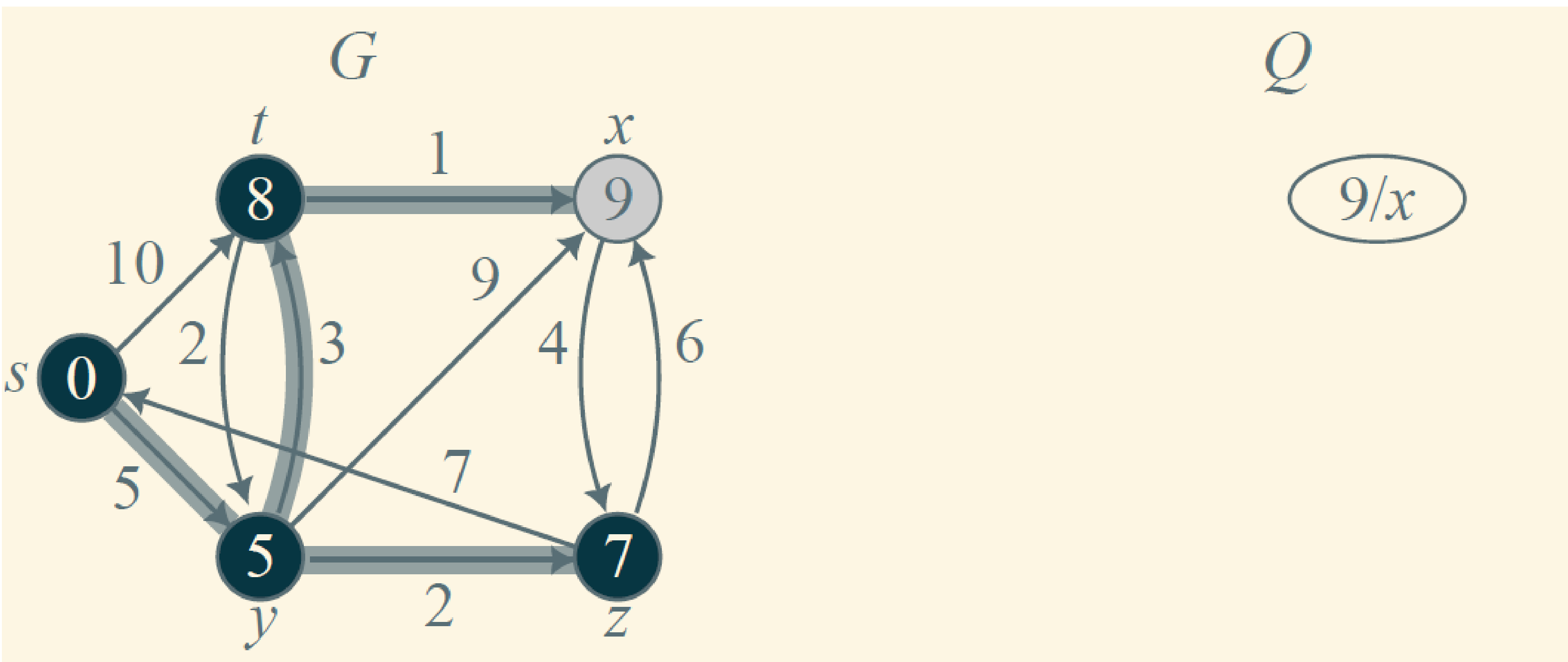
Illustration: Once vertex u with shortest distance is selected, relax for u's connected vertices, greedy method.



Relax edges from z: z → s (7), z → x (6).
Update distances: s done, x = d[z] + 6 = 13 (less than 14).

Dijkstra's Algorithm: Step-by-Step Illustration

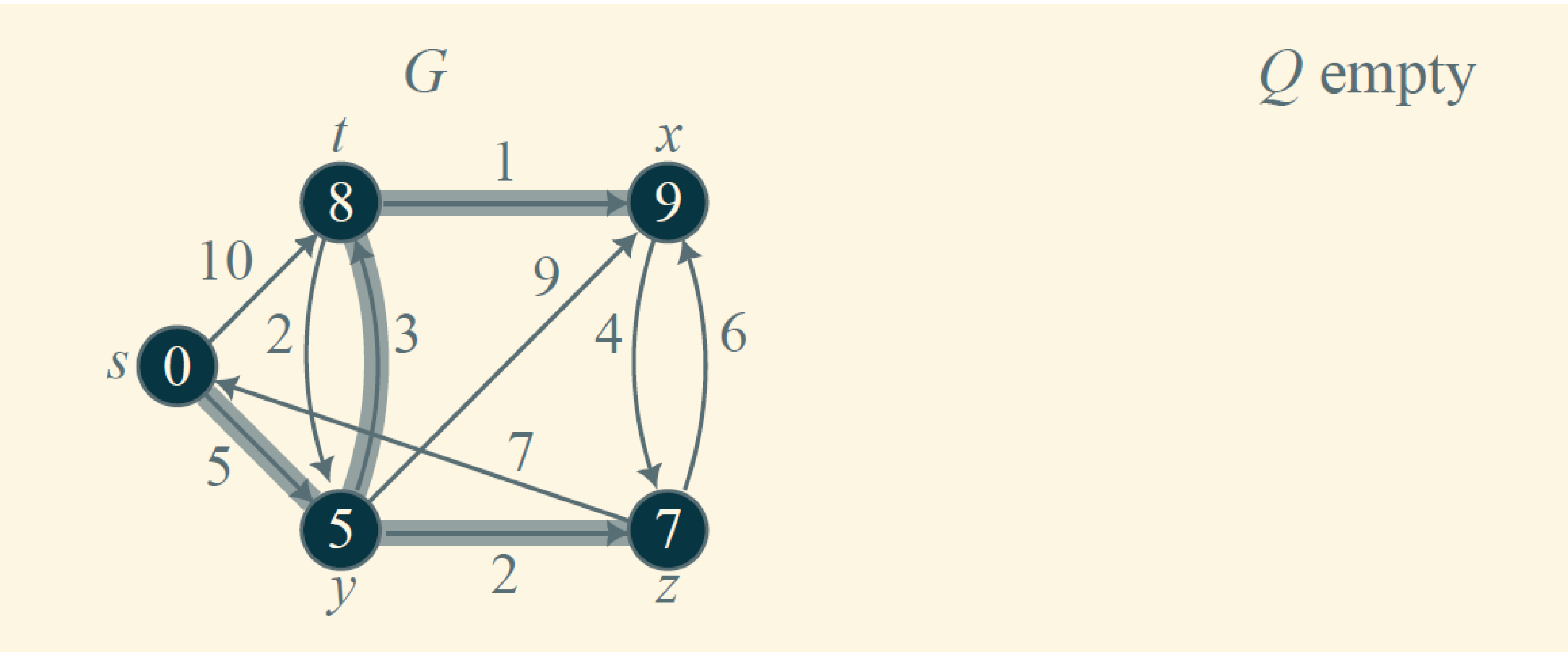
Illustration: Once vertex u with shortest distance is selected, relax for u's connected vertices, greedy method.



Relax edges from t: t → x (1), t → y (2).
Update distances: x = d[t] + 1 = 9 (less than 13), y done.

Dijkstra’s Algorithm: Step-by-Step Illustration

Illustration: Once vertex u with shortest distance is selected, relax for u’s connected vertices, greedy method.



Until the priority queue is empty.

Implementing Dijkstra's Algorithm

- Essentially Dijkstra is a weighted version of BFS.
  - Instead of a FIFO queue, uses a priority queue (heap).
  - Keys are shortest-path weights from source s $(v.d)$.
- Have two sets of vertices:
  - $S =$ vertices whose final shortest-path weights are determined,
  - $Q =$ priority queue = $V - S$.

The algorithm guarantees that once a vertex is extracted from the priority queue, its shortest distance is final (greedy property).

Implementing Dijkstra's Algorithm

Similar as Prim's MST, but relaxing $v.d$ (shortest-path weights from source s) as keys.

```
MST-PRIM(G, w, r)
for each vertex u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)          // add u
    for each vertex v in G.Adj[u]   // update
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            DECREASE-KEY(Q, v, w(u, v))
```

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

Initialization (lines 1–5 of DIJKSTRA)

Relaxation (lines 9–10 of DIJKSTRA)

Implementing Dijkstra's Algorithm

Similar as Prim's MST, but relaxing $v.d$ (shortest-path weights from source s) as keys.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = ∅
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

Initialization

$INIT - SINGLE - SOURCE(G, s)$

$for\ each\ v \in G.V$

$v.d = \infty$ // shortest path estimate from source

$v.\pi = NIL$ // parent/predecessor vertex

$s.d = 0$

Relaxation

$RELAX(u, v, w)$

$if\ v.d > u.d + w(u, v)$

$v.d = u.d + w(u, v)$

$v.\pi = u$

Time Complexity Analysis

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = ∅
    for each vertex u ∈ G.V
        INSERT(Q, u)
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v in G.Adj[u]
            RELAX(u, v, w)
            if the call of RELAX decreased v.d
                DECREASE-KEY(Q, v, v.d)
```

EXTRACT-MIN: Called at most $|V|$ times because, once a vertex is removed from Q, it is never inserted again.

DECREASE-KEY: Called $|E|$ times of all adjacency list elements in a directed graph.

Total time:

If binary heap, each operation takes $O(logV)$, thus total $O((E+V)logV)$.

If connected graph, $E = V-1$ or V^2, equal or larger than V, so $|V| = O(E)$, then total $\boldsymbol{O(E\ logV)}$.

C++ Code Implementation

Implemented using min-heap

Step 1: Construct data structures & key relaxation

C++ STL

```cpp
// Structure to represent a weighted edge in the graph
struct Edge {
    int to, weight;
};

// Type alias for adjacency list representation
typedef vector<vector<Edge>> Graph;

// Relaxation function to update distances
void relax(int u, int v, int weight, vector<int>& dist, vector<int>& parent, priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>>& minHeap) {
    if (dist[u] + weight < dist[v]) {
        dist[v] = dist[u] + weight;
        parent[v] = u;
        minHeap.push({dist[v], v}); // Decrease-Key operation
    }
}
```

// Min-heap stores $\{v.d, v\}$

C++ Code Implementation

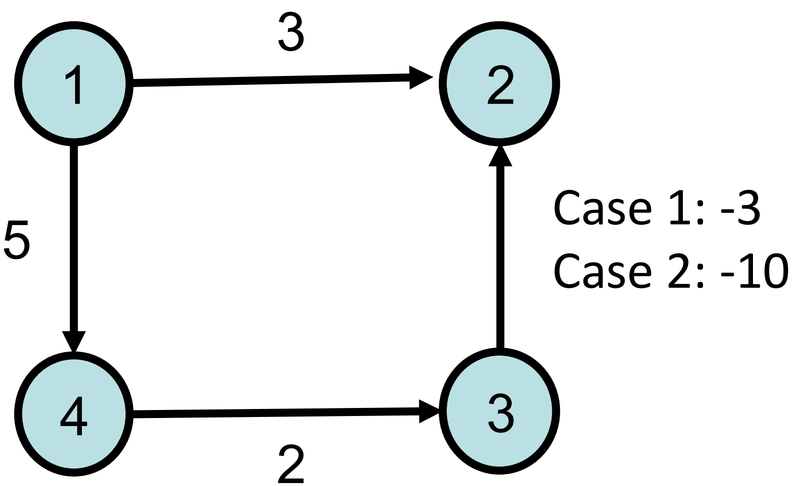
Implemented using min-heap

Step 2: Dijkstra procedure

```cpp
// Dijkstra's Algorithm using Min-Heap
vector<int> dijkstra(const Graph &graph, int source) {
    int V = graph.size(); // Number of vertices
    vector<int> dist(V, INT_MAX); // Distance array initialized to infinity
    vector<int> parent(V, -1); // Parent array for path reconstruction
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap; // Min-Heap

    // Initialize source vertex
    dist[source] = 0;
    minHeap.push({0, source});

    while (!minHeap.empty()) {
        int u = minHeap.top().second;
        minHeap.pop();

        // Process each adjacent vertex
        for (const Edge &edge : graph[u]) {
            int v = edge.to;
            int weight = edge.weight;
            relax(u, v, weight, dist, parent, minHeap);
        }
    }
    return dist;
}
```

Discussion on Dijkstra's Algorithm

- May or may not work for negative-weight edges.
  - For example, find shortest distance from vertex 1 to 2 for below two cases using Dijkstra's algorithm:



Dijkstra's algorithm will provide 1->2 as the shortest path with distance 3, which is incorrect in case 2.

In real-world graphs, a negative weight represents a **benefit, discount, or gain** that reduces the overall cost of a path.

Single-Source Shortest Paths Algorithms

- Bellman-Ford algorithm

Bellman-Ford Algorithm

- Bellman-Ford algorithm finds the shortest paths from a source vertex to all other vertices, and can handle graphs with negative edge weights (but not negative cycles).
- Key differences from Dijkstra's:
  - Bellman-Ford relaxes **all edges** in each iteration, not just from the minimum vertex
  - Requires $(|V| - 1)$ iterations to guarantee shortest paths are found
  - Can detect negative cycles, which Dijkstra's cannot handle

	Complexity	Author
	$O(n^4)$	Shimbel (1955) [30]
	$O(Wn^2m)$	Ford (1956) [14]
*	$O(nm)$	Bellman (1958) [1], Moore (1959) [25]
	$O(n^{\frac{3}{4}}m \log W)$	Gabow (1983) [9]
	$O(\sqrt{n}m \log(nW))$	Gabow and Tarjan (1989) [10]
*	$O(\sqrt{n}m \log(W))$	Goldberg (1993) [12]
*	$\tilde{O}(Wn^\omega)$	Sankowski (2005) [27] Yuster and Zwick (2005) [35]
*	$\tilde{O}(m^{10/7} \log W)$	Cohen, Madry, Sankowski, Vladu (2016)

Table 1: The complexity results for the SSSP problem with negative weights (* indicates asymptotically the best bound for some range of parameters).

Bellman-Ford: Step-by-Step Illustration

HOW IT WORKS:

1. Initialize: Set distance to source $dist[s]$ = 0, all others = ∞.

2. Relaxation Iterations: Repeat $(|V| - 1)$ times:

- For each edge (u, v) with weight w, if $dist[u] + w < dist[v]$, update $dist[v] = dist[u] + w$.

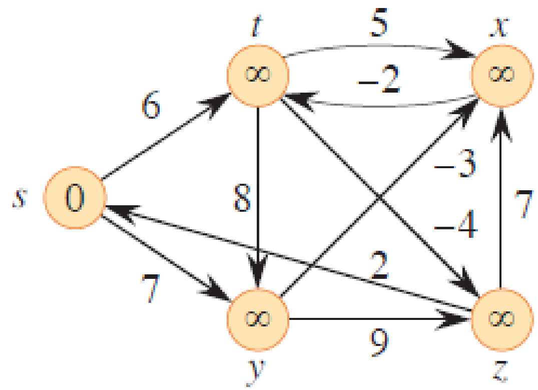
3. Negative Cycle Detection: After $(|V| - 1)$ iterations, check if any edge can still be relaxed.

- If yes, a negative cycle exists.

In a graph with $|V|$ vertices, the longest possible simple path (no repeated vertices) has $(|V| - 1)$ edges.

Bellman-Ford: Step-by-Step Illustration

Illustration: Relax all edges for $|V| - 1$ times, dynamic programming.



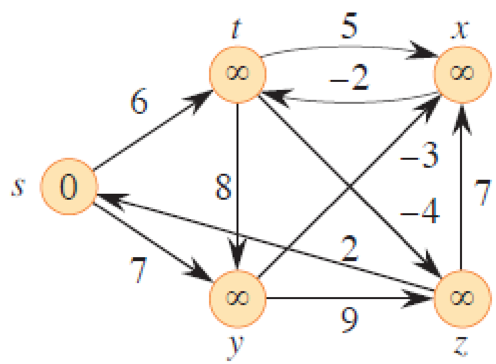
List all edges: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.
Initialization: $d[s] = 0, d[t] = \infty, d[x] = \infty, d[y] = \infty, d[z] = \infty$.

Similar to Dijkstra's, $d[v]$ indicate distance of v to source s.
DP Optimal Substructure: The shortest path to a vertex depends on the shortest paths to previous vertices.

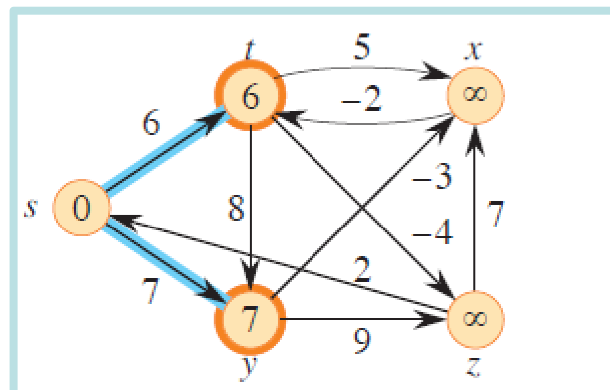
Bellman-Ford: Step-by-Step Illustration

Illustration: Relax all edges for $|V| - 1$ times, dynamic programming.

Edge list: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



(a)



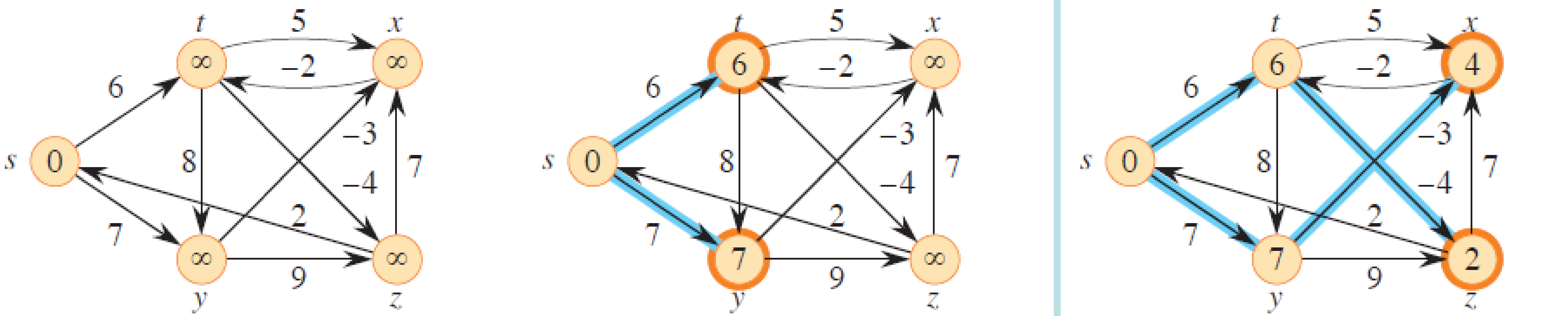
(b) Iteration 1

Iteration 1: Relax all edges in the list one by one, given that $d[s] = 0$:
after $(s, t), (s, y), d[s] = 0, d[t] = 6, d[x] = \infty, d[y] = 7, d[z] = \infty$

Bellman-Ford: Step-by-Step Illustration

Illustration: Relax all edges for $|V| - 1$ times, dynamic programming.

Edge list: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



(a) (b) Iteration 1 (c) Iteration 2

Iteration 2: Relax all edges in the list one by one given iteration 1 results

(t, x): $d[s] = 0, d[t] = 6, d[x] = 11, d[y] = 7, d[z] = \infty$

(t, y): $d[s] = 0, d[t] = 6, d[x] = 11, d[y] = 7, d[z] = \infty$

(t, z): $d[s] = 0, d[t] = 6, d[x] = 11, d[y] = 7, d[z] = 2$

(x, t): $d[s] = 0, d[t] = 6, d[x] = 11, d[y] = 7, d[z] = 2$

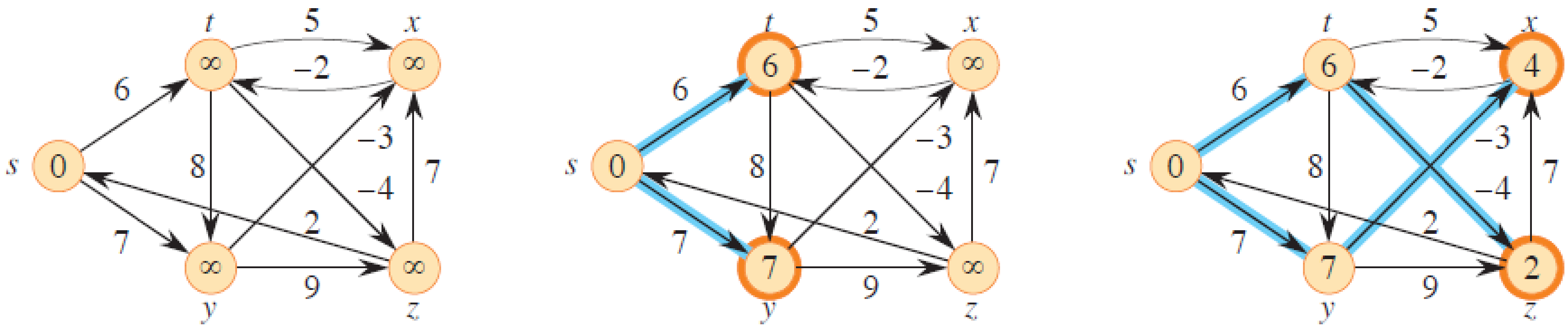
(y, x): $d[s] = 0, d[t] = 6, d[x] = 4, d[y] = 7, d[z] = 2$

$(y, z), (z, x), (z, s), (s, t), (s, y)$: no change

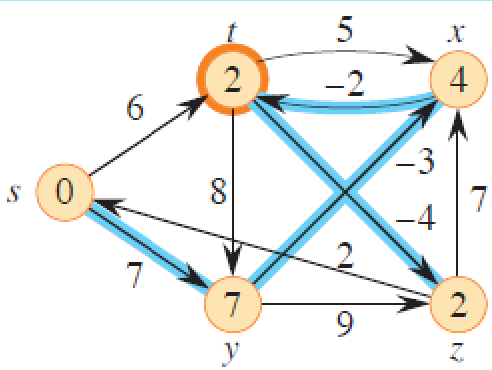
Bellman-Ford: Step-by-Step Illustration

Illustration: Relax all edges for $|V| - 1$ times, dynamic programming.

Edge list: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



(a) (b) Iteration 1 (c) Iteration 2



(d) Iteration 3

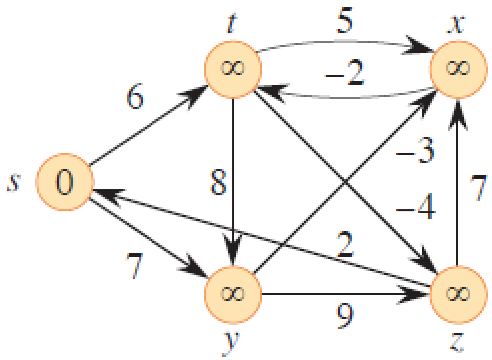
Iteration 3: Relax all edges in the list one by one given iteration 2 results.

- $(t, x), (t, y), (t, z)$: no change
- (x, t): $d[s] = 0, d[t] = 2, d[x] = 4, d[y] = 7, d[z] = 2$
- $(y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$: no change

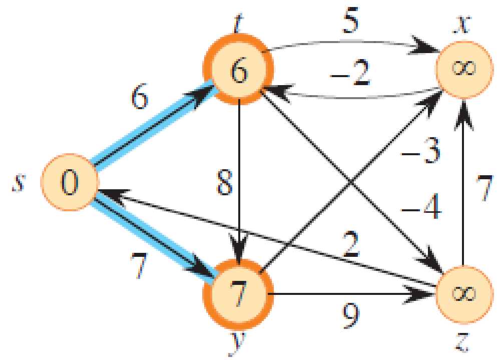
Bellman-Ford: Step-by-Step Illustration

Illustration: Relax all edges for $|V| - 1$ times, dynamic programming.

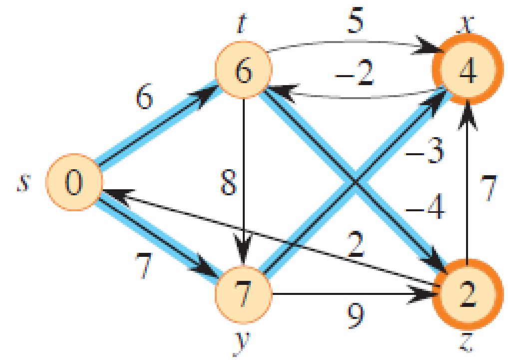
Edge list: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.



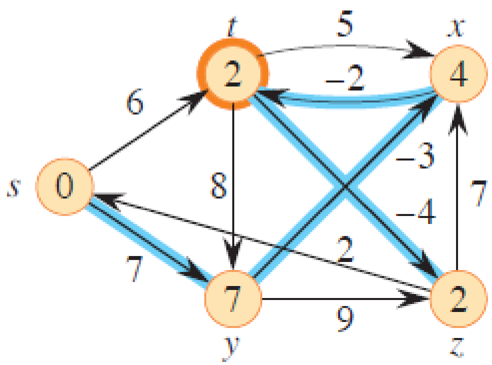
(a)



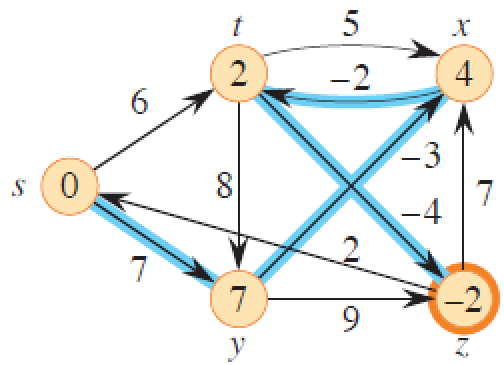
(b) Iteration 1



(c) Iteration 2



(d) Iteration 3

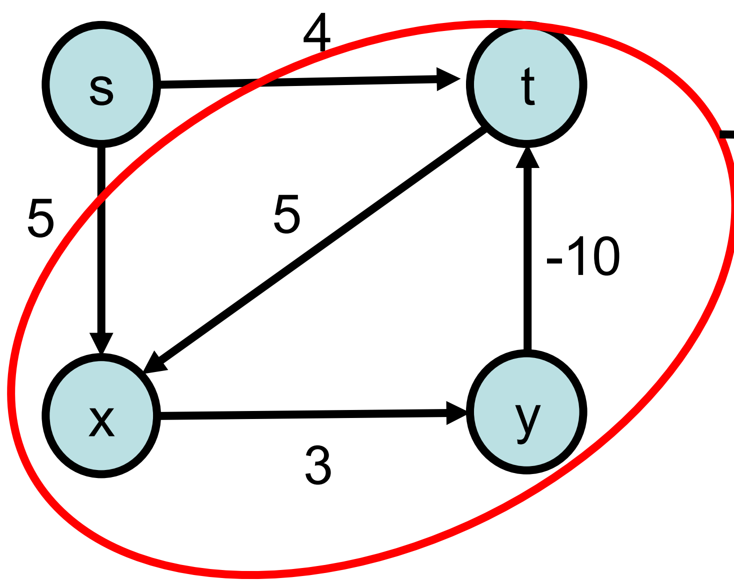


(e) Iteration 4

Iteration 4: With edge (t, z), $d[z]$ relax from 2 to -2; When examining the rest edges, no change.

Discussion on Bellman-Ford Algorithm

After $|V| - 1$ = 3 iterations, what are the shortest path distances from the source s to vertices x, t, and y using Bellman-Ford?



- Fail if there is a negative-weight cycle (no final optimal solution).
- With negative-weight cycle, keep going around it and will get $w(s, v) = -\infty$ for all v on the cycle.

Iter	(s,t)	(s,x)	(t,x)	(x,y)	(y,t)
1		d[x] = 5		d[y] = 8	d[t] = -2
2			d[x] = 3	d[y] = 6	d[t] = -4
3			d[x] = 1	d[y] = 4	d[t] = -6

Discussion on Bellman-Ford Algorithm

- Different edge orders across iterations may
  - propagate distance updates **faster or slower**
  - cause intermediate values to differ
- But after $(|V| - 1)$ **iterations**, the final distances will still be correct (**assuming no negative cycles**).

Implementing Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| - 1
        for each edge (u, v) ∈ G.E
            RELAX(u, v, w)
    for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
            return FALSE
    return TRUE
```

// Core: Relax all edges $|V| - 1$ times.

// After $(|V| - 1)$ iterations, check if any edge can still be relaxed. If yes, a negative cycle exists, return False.

Time Complexity Analysis

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
        for each edge (u, v) ∈ G.E
                RELAX(u, v, w)
for each edge (u, v) ∈ G.E
        if v.d > u.d + w(u, v)
                return FALSE
return TRUE
```

→ Relax all edges $|V| - 1$ times

→ $\Theta(V + E)$ examining $|V|$ adjacent lists to find $|E|$ edges.

Total time:

$O(V^2 + VE)$ because fewer than $|V| - 1$ passes sometimes suffice;
$\Rightarrow \boldsymbol{O(VE)}$ when $|V| = O(E)$ in the frequent case.

Slower than Dijkstra's algorithm but more versatile.

C++ Code Implementation

```
void BellmanFord(Graph& graph, int src) {
    int V = graph.V;
    int E = graph.E;
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    // Relax all edges V-1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph.edges[j].src;
            int v = graph.edges[j].dest;
            int weight = graph.edges[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Check for negative-weight cycles
    for (int i = 0; i < E; i++) {
        int u = graph.edges[i].src;
        int v = graph.edges[i].dest;
        int weight = graph.edges[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains negative weight cycle" << endl;
            return;
        }
    }
```

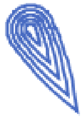
Summary

Single-source shortest path problem and algorithms:

ALGORITHM	SITUATION	TYPE	TIME COMPLEXITY
Breadth-First Search (BFS)	Unweight (w = 1)	Iterative	O(V + E)
Dijkstra's Algorithm	Non-negative weight edges	Greedy	O(E log V)
Bellman-Ford Algorithm	General directed weight edges	Dynamic Programming	O(V * E)

BFS

Dijkstra



A^* (extension)

Summary

- Dijkstra's Algorithm:
  - Greedy approach: always processes the vertex with minimum distance
  - Uses priority queue (min-heap) for efficient vertex selection
  - Cannot handle negative edge weights
  - Optimal for graphs with non-negative weights
- Bellman-Ford Algorithm:
  - Dynamic programming approach: relaxes all edges $(|V| - 1)$ times
  - Can detect negative cycles by checking if distances can still be improved after $(|V| - 1)$ iterations
  - Works with negative edge weights (but not negative cycles)
  - Slower than Dijkstra's but more versatile
- **When to Use Each:**
  - **Use Dijkstra's for: non-negative weights, need for efficiency**
  - **Use Bellman-Ford for: negative weights possible, need to detect negative cycles**

Additional Materials

- Pathfinding Algorithms in Game Development (A* Pathfinding)
- Breaking the Shortest Path Barrier: A Deep Dive into BMSSP
- App demo: https://visualgo.net/en/sssp.
- Full list of materials are organized at NotebookLM.