LECTURE 11 OF 12

Minimum Spanning Trees and Shortest Paths

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne

Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 4.3 & 4.4

Duration: 4 hours (with breaks)

Learning Objectives

1. Understand the edge-weighted graph abstraction, its API, and adjacency-list representation with weighted edges.
2. State and prove the cut property and cycle property for minimum spanning trees, and derive the generic greedy MST algorithm.
3. Implement Kruskal's algorithm using Union-Find with path compression and union by rank, and analyze its time complexity.
4. Implement both the lazy and eager versions of Prim's algorithm using priority queues, and compare their performance characteristics.
5. Implement Dijkstra's single-source shortest-path algorithm, prove its correctness for non-negative edge weights, and understand why negative weights violate its assumptions.
6. Implement the Bellman-Ford algorithm for graphs with negative edge weights, detect negative cycles, and compare the four major graph optimization algorithms studied in this lecture.

Part I: Edge-Weighted Graphs

Estimated time: 15 minutes

1.1 Motivation

In Lectures 9 and 10 we studied unweighted graphs and their traversal algorithms. Many real-world problems, however, attach a numerical cost, distance, or capacity to each edge. A road network assigns distances to road segments; a communication network assigns latencies or bandwidths to links; an airline assigns ticket prices to routes. To model such problems we need **edge-weighted graphs**, the foundational abstraction for this entire lecture.

An edge-weighted graph is simply a graph in which each edge carries a real-valued weight. We will study two fundamental optimization problems on edge-weighted graphs: finding a **minimum spanning tree** (MST) in an undirected graph, and finding **shortest paths** from a single source in a directed graph. Both problems have enormous practical significance and elegant algorithmic solutions grounded in greedy strategies.

1.2 The Edge Abstraction

We begin by defining a weighted edge as a first-class object.

DEFINITION (Weighted Edge). A *weighted edge* in an undirected graph is a triple (v, w, weight) where v and w are vertices and weight is a real number. The edge connects v and w with cost equal to weight. We say v and w are the *endpoints* of the edge, and either endpoint can serve as the "other" vertex given one endpoint.

The companion C++ code represents this as a struct:

```
struct Edge:
    v       : integer       // one endpoint
    w       : integer       // other endpoint
    weight  : real number   // edge weight

    function either() -> integer:
        return v

    function other(vertex) -> integer:
        if vertex == v then return w
        if vertex == w then return v
        error "Inconsistent edge"

    function compareTo(that : Edge) -> integer:
        if this.weight < that.weight then return -1
        if this.weight > that.weight then return +1
        return 0
```

The `either()` and `other()` methods allow us to traverse an edge from either direction, which is essential for undirected-graph algorithms. The `compareTo()` method establishes a natural ordering by weight, which Kruskal's algorithm and Prim's algorithm both exploit through priority queues.

1.3 Edge-Weighted Graph API

An edge-weighted graph maintains a collection of weighted edges organized for efficient adjacency queries.

```
class WeightedGraph:
    V       : integer                   // number of vertices
    E       : integer                   // number of edges
    adj     : array[0..V-1] of List<Edge>   // adjacency lists

    constructor(V):
        this.V = V
        this.E = 0
        for v = 0 to V - 1:
            adj[v] = new empty list

    function addEdge(e : Edge):
        v = e.either()
        w = e.other(v)
        adj[v].add(e)
        adj[w].add(e)       // undirected: add to both lists
        E = E + 1

    function adjacent(v) -> Iterable<Edge>:
        return adj[v]

    function edges() -> Iterable<Edge>:
        bag = new empty list
        for v = 0 to V - 1:
            for each edge e in adj[v]:
                if e.other(v) > v:      // avoid duplicates
                    bag.add(e)
        return bag

    function vertexCount() -> integer:
        return V

    function edgeCount() -> integer:
        return E
```

Key design decisions. Each undirected edge appears in *two* adjacency lists (one for each endpoint). The `edges()` method avoids returning duplicates by only including an edge when iterating from its lower-numbered endpoint. The space complexity is Theta(V + E), identical to the unweighted adjacency-list representation from Lecture 9.

1.4 Representation Trade-offs

For edge-weighted graphs, the adjacency-list representation remains the most practical choice for the algorithms we study. An adjacency matrix would use Theta(V^2) space and make edge iteration slow for sparse graphs. An edge list would make adjacency queries slow. The adjacency-list representation provides:

- **Space:** Theta(V + E)
- **Add edge:** O(1)
- **Iterate over edges adjacent to v:** O(degree(v))
- **Iterate over all edges:** O(V + E)

These bounds are sufficient for all four algorithms in this lecture to achieve their optimal time complexities.

1.5 Directed Weighted Graphs

For shortest-path algorithms (Dijkstra, Bellman-Ford), we need **directed** weighted graphs, also called edge-weighted digraphs. The only structural difference is that `addEdge` places the edge in only one adjacency list (the source vertex), and we speak of the edge as going *from* v *to* w.

```
class WeightedDigraph:
    V   : integer
    E   : integer
    adj : array[0..V-1] of List<DirectedEdge>

    function addEdge(e : DirectedEdge):
        v = e.from()
        adj[v].add(e)
        E = E + 1
```

A `DirectedEdge` has `from()` and `to()` methods rather than the symmetric `either()` / `other()` interface of undirected edges.

Part II: Minimum Spanning Trees -- Definitions and Properties

Estimated time: 20 minutes

2.1 Spanning Trees

DEFINITION (Spanning Tree). Given a connected, undirected graph G = (V, E), a *spanning tree* T is a subgraph of G that is a tree (connected and acyclic) and includes every vertex of G. A spanning tree of a graph with V vertices has exactly V - 1 edges.

A connected graph with V vertices and E edges has many spanning trees in general. For example, the complete graph K_4 has 4^2 = 16 spanning trees (by Cayley's formula, K_n has n^{n-2} spanning trees). The number grows super-exponentially, so exhaustive search is infeasible for finding an optimal one.

2.2 Minimum Spanning Tree

DEFINITION (Minimum Spanning Tree). Given a connected, edge-weighted, undirected graph G = (V, E, w), a *minimum spanning tree* (MST) is a spanning tree T whose total weight

```
w(T) = sum of w(e) for all e in T
```

is minimized over all spanning trees of G. If all edge weights are distinct, the MST is unique.

The MST problem arises in network design (connecting cities with minimum total cable), clustering (removing the longest edges from an MST yields a k-clustering that maximizes the minimum inter-cluster distance), and approximation algorithms for NP-hard problems like the traveling salesman problem.

2.3 The Cut Property

The theoretical foundation of all efficient MST algorithms is the **cut property**.

DEFINITION (Cut). A *cut* of a graph G = (V, E) is a partition of V into two non-empty disjoint sets S and V - S. An edge (v, w) *crosses* the cut if v is in S and w is in V - S (or vice versa). A *crossing edge of minimum weight* is called a *light edge* of the cut.

PROPOSITION A (Cut Property). Let G = (V, E, w) be a connected, edge-weighted, undirected graph. Let S be any subset of V, and let e be the unique minimum-weight edge crossing the cut (S, V - S). Then e belongs to every MST of G.

Proof. Suppose for contradiction that T is an MST that does not contain e. Let e = (u, v) with u in S and v in V - S. Since T is a spanning tree, T contains a unique path P from u to v. This path must cross the cut (S, V - S) at least once, so there exists some edge f in P that crosses the cut. Now f is not equal to e (since e is not in T), and w(f) > w(e) (since e is the unique minimum-weight crossing edge).

Consider the tree T' = T - {f} + {e}. Removing f from T disconnects T into two components; adding e reconnects them (because e also crosses the cut). So T' is a spanning tree. Its weight is:

```
w(T') = w(T) - w(f) + w(e) < w(T)
```

This contradicts the assumption that T is a minimum spanning tree. Therefore e must be in every MST. QED.

When edge weights are not all distinct, the cut property generalizes: *some* minimum-weight crossing edge belongs to *some* MST. This weaker statement is sufficient for algorithmic purposes.

2.4 The Cycle Property

The cycle property is the dual of the cut property and is useful for proving that certain edges are *not* in the MST.

PROPOSITION B (Cycle Property). Let G = (V, E, w) be a connected, edge-weighted, undirected graph. Let C be any cycle in G, and let e be the unique maximum-weight edge in C. Then e does not belong to any MST of G.

Proof. Suppose for contradiction that T is an MST containing e. Removing e from T partitions V into two components, defining a cut (S, V - S). Since C is a cycle containing e, C must contain another edge f that also crosses this cut (a cycle that crosses a cut must cross it an even number of times). We have w(f) < w(e)

because e is the unique maximum-weight edge on the cycle. Then T' = T - {e} + {f} is a spanning tree with w(T') < w(T), contradicting the minimality of T. QED.

2.5 The Greedy MST Algorithm

The cut property directly yields a generic greedy algorithm for MST:

```
function GreedyMST(G):
    T = empty set of edges
    while |T| < V - 1:
        Find a cut (S, V-S) such that no edge in T crosses it
        Add the minimum-weight crossing edge e to T
    return T
```

The correctness of this algorithm follows immediately from the cut property: at each step, the minimum-weight crossing edge must belong to the MST, so we are safe to include it. The algorithm terminates when T has V - 1 edges, at which point T is a spanning tree.

Both Kruskal's and Prim's algorithms are instantiations of this generic greedy strategy. They differ in *how* they choose the cut at each step.

2.6 Number of Edges in an MST

PROPOSITION C. Every spanning tree of a connected graph with V vertices has exactly V - 1 edges.

Proof. By induction on V. Base case: V = 1, the spanning tree has 0 = V - 1 edges. Inductive step: assume every spanning tree on fewer than V vertices has V' - 1 edges. Let T be a spanning tree on V vertices. Since T is a tree, it has at least one leaf (a vertex of degree 1). Remove this leaf and its incident edge. The remaining graph is a spanning tree on V - 1 vertices and, by the inductive hypothesis, has V - 2 edges. So T has V - 2 + 1 = V - 1 edges. QED.

This fact is used in both Kruskal's and Prim's algorithms as a termination condition.

Part III: Kruskal's Algorithm

Estimated time: 30 minutes

3.1 Overview

Kruskal's algorithm (Joseph Kruskal, 1956) is perhaps the most intuitive MST algorithm. The idea is simple: sort all edges by weight, then process them in order, adding each edge to the MST if it does not create a cycle. The cut that justifies each addition is the partition of vertices into the connected components of the forest built so far.

3.2 Algorithm Description

```
function KruskalMST(G):
    mst = empty list               // edges in the MST
    sort all edges of G by weight in non-decreasing order
    uf = new UnionFind(G.V)         // initially V components

    for each edge e = (v, w, weight) in sorted order:
        if uf.find(v) != uf.find(w):    // v and w in different components
            mst.add(e)
            uf.union(v, w)              // merge components
            if |mst| == G.V - 1:
                break                   // MST complete

    return mst
```

Correctness. At each step, the edge e connects two different components. The components define a cut, and since edges are processed in weight order, e is the minimum-weight edge crossing that cut. By the cut property (Proposition A), e belongs to the MST.

3.3 Union-Find Data Structure

Kruskal's algorithm requires a data structure that supports two operations efficiently: `find(v)` (which component contains v?) and `union(v, w)` (merge the components containing v and w). The **Union-Find** (disjoint-set) data structure solves this problem with nearly constant amortized time per operation.

DEFINITION (Union-Find). A *Union-Find* data structure maintains a collection of disjoint sets. Each set is identified by a *representative* element (the root of its tree). The two fundamental operations are:

- `find(x)`: return the representative of the set containing x.
- `union(x, y)`: merge the sets containing x and y into a single set.

```
class UnionFind:
    parent : array[0..N-1] of integer
    rank   : array[0..N-1] of integer
    count  : integer                    // number of components

    constructor(N):
        count = N
        for i = 0 to N - 1:
            parent[i] = i              // each element is its own root
            rank[i] = 0               // all trees have rank 0

    function find(x) -> integer:
        while parent[x] != x:
            parent[x] = parent[parent[x]]   // path compression (halving)
            x = parent[x]
        return x

    function union(x, y):
        rootX = find(x)
        rootY = find(y)
        if rootX == rootY:
            return                     // already in same component

        // Union by rank: attach smaller tree under larger tree
        if rank[rootX] < rank[rootY]:
            parent[rootX] = rootY
        else if rank[rootX] > rank[rootY]:
            parent[rootY] = rootX
        else:
            parent[rootY] = rootX
            rank[rootX] = rank[rootX] + 1

        count = count - 1

    function connected(x, y) -> boolean:
        return find(x) == find(y)

    function componentCount() -> integer:
        return count
```

Path compression (line `parent[x] = parent[parent[x]]`) is the one-pass "halving" variant: each node on the find path is redirected to its grandparent. Full path compression (making every node point directly to the root) achieves the same amortized bound but requires two passes or recursion. Both variants yield amortized time O(alpha(N)) per operation, where alpha is the inverse Ackermann function.

Union by rank ensures that the tree representing each set remains balanced. The rank is an upper bound on the height of the tree rooted at each node. When two trees of different rank are merged, the smaller-rank tree becomes a child of the larger-rank root, preserving the rank. When two trees of equal rank are merged, one becomes a child of the other and the new root's rank increases by 1.

PROPOSITION D. With union by rank and path compression, any sequence of M union/find operations on N elements takes O(M * alpha(N)) time, where alpha is the inverse Ackermann function. For all practical values of N (up to 2^{2^{2^{65536}}}), alpha(N) <= 5, so the amortized cost per operation is effectively O(1).

Proof sketch. The proof, due to Tarjan (1975) and later simplified by Seidel and Sharir (2005), uses an intricate amortized analysis based on a rank-indexed potential function. The key insight is that path compression dramatically flattens the trees over time, and union by rank ensures that ranks grow at most logarithmically. Together these two optimizations interact synergistically to achieve the nearly-constant amortized bound. The full proof is beyond the scope of this lecture but can be found in Tarjan's original paper or in Cormen et al., Chapter 21. QED.

3.4 Detailed Trace

Consider the following edge-weighted graph with 8 vertices (0 through 7) and 16 edges:

```
Edge List (unsorted):
  (0,7,0.16)  (2,3,0.17)  (1,7,0.19)  (0,2,0.26)
  (5,7,0.28)  (1,3,0.29)  (1,5,0.32)  (2,7,0.34)
  (4,5,0.35)  (1,2,0.36)  (4,7,0.37)  (0,4,0.38)
  (6,2,0.40)  (3,6,0.52)  (6,0,0.58)  (6,4,0.93)
```

Step 0: Sort edges by weight.

```
Sorted edges:
  1. (0,7,0.16)    5. (5,7,0.28)    9.  (4,5,0.35)   13. (6,2,0.40)
  2. (2,3,0.17)    6. (1,3,0.29)   10.  (1,2,0.36)   14. (3,6,0.52)
  3. (1,7,0.19)    7. (1,5,0.32)   11.  (4,7,0.37)   15. (6,0,0.58)
  4. (0,2,0.26)    8. (2,7,0.34)   12.  (0,4,0.38)   16. (6,4,0.93)
```

Step 1: Process (0,7,0.16).

- find(0) = 0, find(7) = 7. Different components.
- Add edge to MST. Union(0, 7).
- UF state: {0,7}, {1}, {2}, {3}, {4}, {5}, {6}. Components: 7. MST edges: 1.

Step 2: Process (2,3,0.17).

- find(2) = 2, find(3) = 3. Different components.
- Add edge to MST. Union(2, 3).
- UF state: {0,7}, {1}, {2,3}, {4}, {5}, {6}. Components: 6. MST edges: 2.

Step 3: Process (1,7,0.19).

- find(1) = 1, find(7) = 0 (7's root is 0). Different components.
- Add edge to MST. Union(1, 0).
- UF state: {0,1,7}, {2,3}, {4}, {5}, {6}. Components: 5. MST edges: 3.

Step 4: Process (0,2,0.26).

- find(0) = 0, find(2) = 2. Different components.
- Add edge to MST. Union(0, 2).
- UF state: {0,1,2,3,7}, {4}, {5}, {6}. Components: 4. MST edges: 4.

Step 5: Process (5,7,0.28).

- find(5) = 5, find(7) = 0. Different components.
- Add edge to MST. Union(5, 0).
- UF state: {0,1,2,3,5,7}, {4}, {6}. Components: 3. MST edges: 5.

Step 6: Process (1,3,0.29).

- find(1) = 0, find(3) = 0 (after path compression through 2). **Same component.**
- Skip this edge (would create a cycle).

Step 7: Process (1,5,0.32).

- find(1) = 0, find(5) = 0. **Same component.**
- Skip this edge.

Step 8: Process (2,7,0.34).

- find(2) = 0, find(7) = 0. **Same component.**
- Skip this edge.

Step 9: Process (4,5,0.35).

- find(4) = 4, find(5) = 0. Different components.
- Add edge to MST. Union(4, 0).
- UF state: {0,1,2,3,4,5,7}, {6}. Components: 2. MST edges: 6.

Step 10: Process (1,2,0.36).

- find(1) = 0, find(2) = 0. **Same component.**
- Skip this edge.

Step 11: Process (4,7,0.37).

- find(4) = 0, find(7) = 0. **Same component.**
- Skip this edge.

Step 12: Process (0,4,0.38).

- find(0) = 0, find(4) = 0. **Same component.**
- Skip this edge.

Step 13: Process (6,2,0.40).

- find(6) = 6, find(2) = 0. Different components.
- Add edge to MST. Union(6, 0).
- UF state: {0,1,2,3,4,5,6,7}. Components: 1. MST edges: 7 = V - 1. **Done.**

Final MST edges:

```
(0,7,0.16)  (2,3,0.17)  (1,7,0.19)  (0,2,0.26)
(5,7,0.28)  (4,5,0.35)  (6,2,0.40)

Total MST weight: 0.16 + 0.17 + 0.19 + 0.26 + 0.28 + 0.35 + 0.40 = 1.81
```

The algorithm processed 13 of 16 edges, accepting 7 and rejecting 6.

3.5 Time Complexity Analysis

PROPOSITION E. Kruskal's algorithm computes the MST of a connected, edge-weighted graph with V vertices and E edges in time O(E log E).

Proof. The algorithm consists of three phases:

1. **Sorting:** Sorting E edges by weight takes O(E log E) time using an efficient comparison sort (mergesort or heapsort).

2. **Union-Find operations:** The algorithm performs at most E find operations and at most V - 1 union operations. With path compression and union by rank, each operation takes amortized O(alpha(V)) time, so the total is O(E * alpha(V)).

3. **Overall:** Since alpha(V) <= 5 for all practical V, the Union-Find phase is O(E). The sorting phase dominates, giving total time O(E log E). Since E <= V^2, we have log E <= 2 log V, so O(E log E) = O(E log V). QED.

Space complexity: O(V + E) for the graph, sorted edge array, Union-Find arrays, and MST edge list.

3.6 When to Use Kruskal's

Kruskal's algorithm is particularly well-suited when:

- The graph is sparse (E is close to V), because the sorting step is cheap.
- The edges are already sorted or nearly sorted, reducing the sorting cost.
- A Union-Find data structure is already available or useful for other purposes.
- The graph is given as an edge list rather than an adjacency list.

For dense graphs (E close to V^2), Prim's algorithm with an adjacency-list representation is often preferable.

Part IV: Prim's Algorithm

Estimated time: 30 minutes

4.1 Overview

Prim's algorithm (Robert Prim, 1957; also discovered independently by Vojtech Jarnik in 1930) grows the MST one vertex at a time, starting from an arbitrary source vertex. At each step, it adds the minimum-weight edge that connects a vertex in the growing tree to a vertex not yet in the tree. The cut that justifies each addition is the partition (tree vertices, non-tree vertices).

The key difference from Kruskal's is the choice of cut: Kruskal's uses a different cut for each edge (defined by the current connected components), while Prim's always uses the same evolving cut (tree vs. non-tree).

4.2 Lazy Prim's Algorithm

The "lazy" version of Prim's algorithm uses a standard min-priority queue to hold all edges that cross the cut. When we add a vertex to the tree, we insert all its adjacent edges into the priority queue. When we extract the minimum edge, we check whether it still crosses the cut (it might not, if both endpoints have since been added to the tree).

```
function LazyPrimMST(G):
    marked = array[0..V-1] of boolean, all false   // in the tree?
    mst    = empty list                             // MST edges
    pq     = new MinPriorityQueue<Edge>             // crossing edges
    totalWeight = 0.0

    // Start from vertex 0
    visit(G, 0, marked, pq)

    while pq is not empty and |mst| < V - 1:
        e = pq.extractMin()             // minimum-weight crossing edge
        v = e.either()
        w = e.other(v)

        if marked[v] and marked[w]:
            continue                    // both endpoints in tree; skip (lazy)

        mst.add(e)                      // add edge to MST
        totalWeight = totalWeight + e.weight

        if not marked[v]:
            visit(G, v, marked, pq)
        if not marked[w]:
            visit(G, w, marked, pq)

    return mst

function visit(G, v, marked, pq):
    marked[v] = true
    for each edge e in G.adjacent(v):
        if not marked[e.other(v)]:
            pq.insert(e)
```

The "lazy" aspect: When a vertex w transitions from non-tree to tree, some edges in the priority queue that previously crossed the cut no longer do. Rather than removing these stale edges (which would require searching the priority queue), we simply leave them and check at extraction time. This is simple but means the priority queue can hold up to E edges.

4.3 Eager Prim's Algorithm

The "eager" version eliminates stale edges by maintaining, for each non-tree vertex w, only the minimum-weight edge connecting w to the tree. This requires an **indexed priority queue** (also called a decrease-key priority queue) that supports efficient updates.

```
function EagerPrimMST(G):
    marked  = array[0..V-1] of boolean, all false
    distTo  = array[0..V-1] of real, all +infinity    // weight of best edge to tree
    edgeTo  = array[0..V-1] of Edge, all null         // best edge connecting to tree
    mst     = empty list
    ipq     = new IndexedMinPQ<real>(V)                // indexed by vertex

    // Start from vertex 0
    distTo[0] = 0.0
    ipq.insert(0, 0.0)

    while ipq is not empty:
        v = ipq.extractMin()           // vertex with minimum distTo
        marked[v] = true

        if edgeTo[v] is not null:
            mst.add(edgeTo[v])

        for each edge e in G.adjacent(v):
            w = e.other(v)
            if marked[w]:
                continue               // w already in tree

            if e.weight < distTo[w]:
                distTo[w] = e.weight
                edgeTo[w] = e

                if ipq.contains(w):
                    ipq.decreaseKey(w, e.weight)
                else:
                    ipq.insert(w, e.weight)

    return mst
```

Indexed priority queue. An indexed min-priority queue associates each index (vertex) with a priority (weight). It supports `insert(index, priority)`, `extractMin()` (returns the index with smallest priority), `decreaseKey(index, newPriority)`, and `contains(index)`, all in O(log N) time. A binary heap augmented with an index-to-position array and a position-to-index array achieves these bounds.

```
class IndexedMinPQ:
    N       : integer                   // maximum number of elements
    n       : integer                   // current number of elements
    keys    : array[0..N-1] of real     // keys[i] = priority of index i
    pq      : array[0..N] of integer    // binary heap (1-indexed)
    qp      : array[0..N-1] of integer  // inverse: qp[i] = position of i in pq
                                        // qp[i] = -1 if i is not in the queue

    function insert(i, key):
        n = n + 1
        qp[i] = n
        pq[n] = i
        keys[i] = key
        swim(n)

    function extractMin() -> integer:
        min = pq[1]
        swap(1, n)
        n = n - 1
        sink(1)
        qp[min] = -1
        return min

    function decreaseKey(i, key):
        keys[i] = key
        swim(qp[i])

    function contains(i) -> boolean:
        return qp[i] != -1

    // swim and sink are standard binary heap operations
    // adjusted to maintain the qp inverse mapping
```

4.4 Detailed Trace (Eager Prim's)

We trace the eager version on the same graph used for Kruskal's trace. Start from vertex 0.

```
Initial state:
  distTo = [0.0, inf, inf, inf, inf, inf, inf, inf]
  edgeTo = [null, null, null, null, null, null, null, null]
  IPQ = {0: 0.0}
```

Step 1: Extract vertex 0 (distTo = 0.0). Mark 0.

- edgeTo[0] = null, so no edge added to MST.
- Process adjacency of 0: edges (0,7,0.16), (0,2,0.26), (0,4,0.38), (0,6,0.58).
- Vertex 7: 0.16 < inf. distTo[7] = 0.16, edgeTo[7] = (0,7,0.16). Insert 7 into IPQ.
- Vertex 2: 0.26 < inf. distTo[2] = 0.26, edgeTo[2] = (0,2,0.26). Insert 2 into IPQ.
- Vertex 4: 0.38 < inf. distTo[4] = 0.38, edgeTo[4] = (0,4,0.38). Insert 4 into IPQ.
- Vertex 6: 0.58 < inf. distTo[6] = 0.58, edgeTo[6] = (6,0,0.58). Insert 6 into IPQ.
- IPQ = {7: 0.16, 2: 0.26, 4: 0.38, 6: 0.58}

Step 2: Extract vertex 7 (distTo = 0.16). Mark 7.

- edgeTo[7] = (0,7,0.16). Add to MST. MST weight = 0.16.
- Process adjacency of 7: edges (0,7,0.16), (1,7,0.19), (5,7,0.28), (2,7,0.34), (4,7,0.37).
- Vertex 0: marked. Skip.
- Vertex 1: 0.19 < inf. distTo[1] = 0.19, edgeTo[1] = (1,7,0.19). Insert 1 into IPQ.
- Vertex 5: 0.28 < inf. distTo[5] = 0.28, edgeTo[5] = (5,7,0.28). Insert 5 into IPQ.
- Vertex 2: 0.34 > 0.26. No update (current edge to 2 is cheaper).
- Vertex 4: 0.37 < 0.38. distTo[4] = 0.37, edgeTo[4] = (4,7,0.37). DecreaseKey 4 in IPQ.
- IPQ = {1: 0.19, 2: 0.26, 5: 0.28, 4: 0.37, 6: 0.58}

Step 3: Extract vertex 1 (distTo = 0.19). Mark 1.

- edgeTo[1] = (1,7,0.19). Add to MST. MST weight = 0.35.
- Process adjacency of 1: edges (1,7,0.19), (1,3,0.29), (1,5,0.32), (1,2,0.36).
- Vertex 7: marked. Skip.
- Vertex 3: 0.29 < inf. distTo[3] = 0.29, edgeTo[3] = (1,3,0.29). Insert 3 into IPQ.
- Vertex 5: 0.32 > 0.28. No update.
- Vertex 2: 0.36 > 0.26. No update.
- IPQ = {2: 0.26, 5: 0.28, 3: 0.29, 4: 0.37, 6: 0.58}

Step 4: Extract vertex 2 (distTo = 0.26). Mark 2.

- edgeTo[2] = (0,2,0.26). Add to MST. MST weight = 0.61.
- Process adjacency of 2: edges (2,3,0.17), (0,2,0.26), (2,7,0.34), (1,2,0.36), (6,2,0.40).

- Vertex 3: 0.17 < 0.29. distTo[3] = 0.17, edgeTo[3] = (2,3,0.17). DecreaseKey 3 in IPQ.
- Vertex 0: marked. Skip.
- Vertex 7: marked. Skip.
- Vertex 1: marked. Skip.
- Vertex 6: 0.40 < 0.58. distTo[6] = 0.40, edgeTo[6] = (6,2,0.40). DecreaseKey 6 in IPQ.
- IPQ = {3: 0.17, 5: 0.28, 4: 0.37, 6: 0.40}

Step 5: Extract vertex 3 (distTo = 0.17). Mark 3.
- edgeTo[3] = (2,3,0.17). Add to MST. MST weight = 0.78.
- Process adjacency of 3: edges (2,3,0.17), (1,3,0.29), (3,6,0.52).
- Vertex 2: marked. Skip.
- Vertex 1: marked. Skip.
- Vertex 6: 0.52 > 0.40. No update.
- IPQ = {5: 0.28, 4: 0.37, 6: 0.40}

Step 6: Extract vertex 5 (distTo = 0.28). Mark 5.
- edgeTo[5] = (5,7,0.28). Add to MST. MST weight = 1.06.
- Process adjacency of 5: edges (5,7,0.28), (1,5,0.32), (4,5,0.35).
- Vertex 7: marked. Skip.
- Vertex 1: marked. Skip.
- Vertex 4: 0.35 < 0.37. distTo[4] = 0.35, edgeTo[4] = (4,5,0.35). DecreaseKey 4 in IPQ.
- IPQ = {4: 0.35, 6: 0.40}

Step 7: Extract vertex 4 (distTo = 0.35). Mark 4.
- edgeTo[4] = (4,5,0.35). Add to MST. MST weight = 1.41.
- Process adjacency of 4: edges (4,5,0.35), (4,7,0.37), (0,4,0.38), (6,4,0.93).
- Vertex 5: marked. Skip.
- Vertex 7: marked. Skip.
- Vertex 0: marked. Skip.
- Vertex 6: 0.93 > 0.40. No update.
- IPQ = {6: 0.40}

Step 8: Extract vertex 6 (distTo = 0.40). Mark 6.
- edgeTo[6] = (6,2,0.40). Add to MST. MST weight = 1.81.
- All neighbors of 6 are marked. IPQ is empty.

Final MST edges (Prim's):

```
(0,7,0.16)  (1,7,0.19)  (0,2,0.26)  (2,3,0.17)
(5,7,0.28)  (4,5,0.35)  (6,2,0.40)

Total MST weight: 1.81
```

Note that Prim's algorithm found the same MST as Kruskal's (as it must, since all edge weights are distinct), but the edges were added in a different order, reflecting the different cut-selection strategy.

4.5 Time Complexity Analysis

PROPOSITION F (Lazy Prim's). The lazy version of Prim's algorithm computes the MST in time O(E log E) and space O(E).

Proof. Each edge is inserted into the priority queue at most twice (once from each endpoint), and each extraction takes O(log E) time. There are at most 2E insertions and at most 2E extractions, giving total time O(E log E). The priority queue holds at most E edges, so space is O(E). QED.

PROPOSITION G (Eager Prim's). The eager version of Prim's algorithm computes the MST in time O(E log V) and space O(V).

Proof. The indexed priority queue holds at most V entries (one per non-tree vertex). Each of the V extractMin operations takes O(log V) time. Each edge is examined once during adjacency scanning, and each examination may trigger an insert or decreaseKey, both O(log V). There are at most E such examinations. Total time is O(V log V + E log V) = O(E log V) since E >= V - 1 for connected graphs. Space is O(V) for the indexed priority queue and the distTo/edgeTo arrays. QED.

The eager version is preferable for dense graphs because log V < log E and the priority queue is smaller.

4.6 Comparison: Kruskal vs. Prim

Property	Kruskal's	Prim's (Eager)
Strategy	Process edges globally by weight	Grow tree from a source vertex
Data structure	Union-Find	Indexed priority queue
Cut selection	Connected components	Tree vs. non-tree
Time complexity	O(E log E)	O(E log V)
Space complexity	O(V + E)	O(V) extra
Best for	Sparse graphs, edge lists	Dense graphs, adjacency lists
Edge processing	All edges (sorted)	Only adjacent edges

Both algorithms are efficient in practice. For very dense graphs, Prim's with a Fibonacci heap achieves O(E + V log V), which is optimal for graphs where E = Omega(V log V).

Part V: Dijkstra's Shortest-Path Algorithm

Estimated time: 35 minutes

5.1 The Shortest-Path Problem

DEFINITION (Shortest Path). Given an edge-weighted digraph G = (V, E, w) with non-negative weights and a source vertex s, the *single-source shortest-path problem* asks for the shortest path from s to every other vertex v. The *shortest-path weight* from s to v is:

```
delta(s, v) = min { w(p) : p is a path from s to v }
```

where w(p) is the sum of edge weights along path p. If no path exists, delta(s, v) = +infinity.

The shortest-path problem is ubiquitous: GPS navigation, network routing (OSPF, IS-IS), scheduling, and many optimization problems reduce to finding shortest paths.

5.2 Shortest-Path Trees

DEFINITION (Shortest-Path Tree). Given a digraph G and source s, a *shortest-path tree* (SPT) rooted at s is a subgraph containing s and all vertices reachable from s, such that the unique path from s to each vertex v in the subgraph is the shortest path from s to v in G.

An SPT has at most V - 1 edges and can be represented compactly by an `edgeTo[]` array, where `edgeTo[v]` is the last edge on the shortest path from s to v. This representation allows path reconstruction by following `edgeTo[]` pointers backward from any vertex to s.

5.3 Edge Relaxation

The fundamental operation in all shortest-path algorithms is **relaxation**.

DEFINITION (Relaxation). Given a directed edge e from v to w with weight w(e), *relaxing* e means testing whether the shortest known path to w can be improved by going through v:

```
function relax(e):
    v = e.from()
    w = e.to()
    if distTo[w] > distTo[v] + e.weight:
        distTo[w] = distTo[v] + e.weight
        edgeTo[w] = e
        return true     // relaxation was successful
    return false
```

If `distTo[v] + e.weight < distTo[w]`, we have found a shorter path to w (via v), so we update `distTo[w]` and record the edge. The name "relaxation" comes from the analogy of a rubber band stretched along a path: replacing a longer subpath with a shorter one "relaxes" the tension.

Optimality conditions. The `distTo[]` values represent shortest-path weights if and only if for every edge (v, w): `distTo[w] <= distTo[v] + w(v,w)`. This is the *shortest-path optimality condition*, and it can be verified in O(E) time.

5.4 Dijkstra's Algorithm

Edsger Dijkstra's algorithm (1959) solves the single-source shortest-path problem for graphs with non-negative edge weights. It is a greedy algorithm that processes vertices in order of their distance from the source, similar to how Prim's algorithm processes vertices in order of their distance to the tree.

```
function Dijkstra(G, s):
    distTo = array[0..V-1] of real, all +infinity
    edgeTo = array[0..V-1] of DirectedEdge, all null
    marked = array[0..V-1] of boolean, all false
    ipq    = new IndexedMinPQ<real>(V)

    distTo[s] = 0.0
    ipq.insert(s, 0.0)

    while ipq is not empty:
        v = ipq.extractMin()
        marked[v] = true

        for each directed edge e from v:
            w = e.to()
            if marked[w]:
                continue            // already settled

            if distTo[v] + e.weight < distTo[w]:
                distTo[w] = distTo[v] + e.weight
                edgeTo[w] = e

                if ipq.contains(w):
                    ipq.decreaseKey(w, distTo[w])
                else:
                    ipq.insert(w, distTo[w])

    return (distTo, edgeTo)
```

Path reconstruction. To recover the actual shortest path from s to a target vertex t:

```
function pathTo(t, edgeTo, s) -> list of edges:
    if distTo[t] == +infinity:
        return null                    // no path exists

    path = new empty stack
    e = edgeTo[t]
    while e is not null:
        path.push(e)
        e = edgeTo[e.from()]

    return path    // edges from s to t (pop for correct order)
```

5.5 Detailed Trace

Consider the following directed weighted graph with 8 vertices (0 through 7). Source vertex is 0.

```
Directed edges:
  0 -> 1 (5.0)    0 -> 4 (9.0)    0 -> 7 (8.0)
  1 -> 2 (12.0)   1 -> 3 (15.0)   1 -> 7 (4.0)
  2 -> 3 (3.0)    2 -> 6 (11.0)
  3 -> 6 (9.0)
  4 -> 5 (4.0)    4 -> 6 (20.0)   4 -> 7 (5.0)
  5 -> 2 (1.0)    5 -> 6 (13.0)
  7 -> 5 (6.0)    7 -> 2 (7.0)
```

Initial state:

```
distTo = [0.0, inf, inf, inf, inf, inf, inf, inf]
edgeTo = [null, null, null, null, null, null, null, null]
IPQ = {0: 0.0}
```

Step 1: Extract vertex 0 (distTo = 0.0). Mark 0.

- Relax 0->1: distTo[1] = 0 + 5 = 5.0. edgeTo[1] = 0->1. Insert 1.
- Relax 0->4: distTo[4] = 0 + 9 = 9.0. edgeTo[4] = 0->4. Insert 4.
- Relax 0->7: distTo[7] = 0 + 8 = 8.0. edgeTo[7] = 0->7. Insert 7.
- IPQ = {1: 5.0, 7: 8.0, 4: 9.0}

```
distTo = [0.0, 5.0, inf, inf, 9.0, inf, inf, 8.0]
```

Step 2: Extract vertex 1 (distTo = 5.0). Mark 1.

- Relax 1->2: distTo[2] = 5 + 12 = 17.0. edgeTo[2] = 1->2. Insert 2.
- Relax 1->3: distTo[3] = 5 + 15 = 20.0. edgeTo[3] = 1->3. Insert 3.
- Relax 1->7: 5 + 4 = 9.0 > 8.0. No update.
- IPQ = {7: 8.0, 4: 9.0, 2: 17.0, 3: 20.0}

```
distTo = [0.0, 5.0, 17.0, 20.0, 9.0, inf, inf, 8.0]
```

Step 3: Extract vertex 7 (distTo = 8.0). Mark 7.

- Relax 7->5: distTo[5] = 8 + 6 = 14.0. edgeTo[5] = 7->5. Insert 5.
- Relax 7->2: 8 + 7 = 15.0 < 17.0. distTo[2] = 15.0. edgeTo[2] = 7->2. DecreaseKey 2.
- IPQ = {4: 9.0, 5: 14.0, 2: 15.0, 3: 20.0}

```
distTo = [0.0, 5.0, 15.0, 20.0, 9.0, 14.0, inf, 8.0]
```

Step 4: Extract vertex 4 (distTo = 9.0). Mark 4.

- Relax 4->5: 9 + 4 = 13.0 < 14.0. distTo[5] = 13.0. edgeTo[5] = 4->5. DecreaseKey 5.
- Relax 4->6: distTo[6] = 9 + 20 = 29.0. edgeTo[6] = 4->6. Insert 6.
- Relax 4->7: marked. Skip.
- IPQ = {5: 13.0, 2: 15.0, 3: 20.0, 6: 29.0}

```
distTo = [0.0, 5.0, 15.0, 20.0, 9.0, 13.0, 29.0, 8.0]
```

Step 5: Extract vertex 5 (distTo = 13.0). Mark 5.

- Relax 5->2: 13 + 1 = 14.0 < 15.0. distTo[2] = 14.0. edgeTo[2] = 5->2. DecreaseKey 2.
- Relax 5->6: 13 + 13 = 26.0 < 29.0. distTo[6] = 26.0. edgeTo[6] = 5->6. DecreaseKey 6.
- IPQ = {2: 14.0, 3: 20.0, 6: 26.0}

```
distTo = [0.0, 5.0, 14.0, 20.0, 9.0, 13.0, 26.0, 8.0]
```

Step 6: Extract vertex 2 (distTo = 14.0). Mark 2.

- Relax 2->3: 14 + 3 = 17.0 < 20.0. distTo[3] = 17.0. edgeTo[3] = 2->3. DecreaseKey 3.

- Relax 2->6: 14 + 11 = 25.0 < 26.0. distTo[6] = 25.0. edgeTo[6] = 2->6. DecreaseKey 6.
- IPQ = {3: 17.0, 6: 25.0}

```
distTo = [0.0, 5.0, 14.0, 17.0, 9.0, 13.0, 25.0, 8.0]
```

Step 7: Extract vertex 3 (distTo = 17.0). Mark 3.
- Relax 3->6: 17 + 9 = 26.0 > 25.0. No update.
- IPQ = {6: 25.0}

```
distTo = [0.0, 5.0, 14.0, 17.0, 9.0, 13.0, 25.0, 8.0]
```

Step 8: Extract vertex 6 (distTo = 25.0). Mark 6.
- No outgoing edges from 6. IPQ is empty.

Final shortest-path distances from vertex 0:

```
Vertex:   0     1     2     3     4     5     6     7
distTo: 0.0   5.0  14.0  17.0   9.0  13.0  25.0   8.0
```

Shortest-path tree (edgeTo reconstruction):

```
Vertex  edgeTo     Path from 0
  1     0->1       0 -> 1
  2     5->2       0 -> 4 -> 5 -> 2
  3     2->3       0 -> 4 -> 5 -> 2 -> 3
  4     0->4       0 -> 4
  5     4->5       0 -> 4 -> 5
  6     2->6       0 -> 4 -> 5 -> 2 -> 6
  7     0->7       0 -> 7
```

5.6 Correctness Proof

PROPOSITION H. Dijkstra's algorithm correctly computes shortest-path distances from source s in an edge-weighted digraph with non-negative edge weights.

Proof. We prove that when a vertex v is extracted from the priority queue, `distTo[v] = delta(s, v)` (the true shortest-path distance). The proof is by strong induction on the extraction order.

Base case: The first vertex extracted is s, with `distTo[s] = 0.0 = delta(s, s)`.

Inductive hypothesis: Assume that for all vertices extracted before v, `distTo[u] = delta(s, u)`.

Inductive step: Suppose for contradiction that when v is extracted, `distTo[v] > delta(s, v)`. Let P be a true shortest path from s to v. Let (x, y) be the first edge on P such that x has been extracted but y has not. (Such an edge must exist because s has been extracted and v has not yet been "settled" with the correct distance.)

Since x was extracted before v, by the inductive hypothesis, `distTo[x] = delta(s, x)`. When x was extracted, edge (x, y) was relaxed, so:

```
distTo[y] <= distTo[x] + w(x, y) = delta(s, x) + w(x, y) = delta(s, y)
```

The last equality holds because the subpath of P from s to y through x is itself a shortest path to y (subpath optimality of shortest paths).

Since y is on the path P from s to v, and all remaining edges from y to v have non-negative weights:

```
delta(s, y) <= delta(s, v)
```

Since v was extracted before y (v has the smallest distTo among unextracted vertices), and both are in the priority queue:

```
distTo[v] <= distTo[y]
```

Combining these inequalities:

```
distTo[v] <= distTo[y] <= delta(s, y) <= delta(s, v)
```

But we assumed `distTo[v] > delta(s, v)`, a contradiction. Therefore `distTo[v] = delta(s, v)`. QED.

Critical requirement: non-negative weights. The step `delta(s, y) <= delta(s, v)` relies on the fact that extending a path by edges with non-negative weights cannot decrease its total weight. With negative edges, this step fails, and the proof breaks down. We explore this next.

5.7 Why Negative Weights Break Dijkstra

Consider the following simple example:

```
Vertices: s, a, b
Edges: s->a (1), s->b (5), a->b (-10)
```

Dijkstra processes vertices in order of distance. It extracts s (distance 0), then a (distance 1), then b (distance 5). When a is extracted, it relaxes a->b: `distTo[b] = min(5, 1 + (-10)) = -9`. But b may have already been marked as settled with distance 5 (in some implementations). Even in implementations that allow updates, the problem is fundamental: Dijkstra assumes that once a vertex is extracted, its distance is final. With negative edges, a later extraction can reveal a shorter path through a negative-weight edge, violating this invariant.

In the example above, the shortest path from s to b is s->a->b with weight 1 + (-10) = -9, but Dijkstra (depending on implementation details) may report 5 or fail to settle b correctly because it processes b before discovering the shorter path through a.

The issue is precisely that the inequality `delta(s, y) <= delta(s, v)` used in the correctness proof no longer holds when edges can have negative weights.

5.8 Time Complexity Analysis

PROPOSITION I. Dijkstra's algorithm runs in O(E log V) time using a binary-heap indexed priority queue, and O(V + E) space.

Proof. The indexed priority queue holds at most V elements. Each vertex is extracted at most once, taking O(log V) per extraction, for a total of O(V log V). Each edge is relaxed at most once; each successful relaxation involves either an insert or a decreaseKey operation, each taking O(log V), for a total of O(E log V). The overall time is O(V log V + E log V) = O(E log V) since E >= V - 1 for connected graphs. Space is O(V) for the priority queue and arrays, plus O(V + E) for the graph. QED.

With a Fibonacci heap, the time improves to O(E + V log V), which is better for dense graphs. However, Fibonacci heaps have large constant factors and are rarely used in practice. For most applications, the binary-heap version is the implementation of choice.

5.9 Variants and Extensions

Shortest paths in undirected graphs. Dijkstra's algorithm works on undirected graphs by treating each undirected edge {v, w} as two directed edges v->w and w->v. This is valid as long as all edge weights are non-negative.

A* search. For point-to-point shortest paths (from s to a specific target t), the A *algorithm uses a heuristic function h(v) that estimates the distance from v to t. A* processes vertices in order of `distTo[v] + h(v)`, which can dramatically reduce the number of vertices explored. With an admissible heuristic (one that never overestimates), A* is guaranteed to find the shortest path.

Dijkstra on DAGs. If the graph is a directed acyclic graph, shortest paths can be found in O(V + E) time by processing vertices in topological order, without a priority queue.

Part VI: Bellman-Ford Algorithm

Estimated time: 20 minutes

6.1 Motivation

Dijkstra's algorithm requires non-negative edge weights. Many real-world problems, however, involve negative weights. Currency exchange arbitrage involves logarithmic edge weights that can be negative; scheduling with deadlines involves negative weights to model penalties; and some network protocols must handle negative costs. The Bellman-Ford algorithm (Richard Bellman, 1958; Lester Ford Jr., 1956) handles all of these cases.

6.2 Algorithm Description

The Bellman-Ford algorithm is conceptually simple: relax *every* edge, and repeat this V - 1 times. After V - 1 passes, the shortest-path distances are guaranteed to be correct (assuming no negative cycles are reachable from the source).

```
function BellmanFord(G, s):
    distTo = array[0..V-1] of real, all +infinity
    edgeTo = array[0..V-1] of DirectedEdge, all null
    distTo[s] = 0.0

    // Relax all edges V - 1 times
    for i = 1 to V - 1:
        for each directed edge e = (v, w, weight) in G:
            if distTo[v] + weight < distTo[w]:
                distTo[w] = distTo[v] + weight
                edgeTo[w] = e

    // Check for negative cycles (V-th pass)
    for each directed edge e = (v, w, weight) in G:
        if distTo[v] + weight < distTo[w]:
            report "Negative cycle detected"
            return null

    return (distTo, edgeTo)
```

6.3 Why V - 1 Passes Suffice

PROPOSITION J. If G has no negative-weight cycles reachable from s, then after V - 1 passes of relaxing all edges, `distTo[v] = delta(s, v)` for every vertex v reachable from s.

Proof. Consider any shortest path from s to v: s = v_0, v_1, v_2, ..., v_k = v. This path has at most V - 1 edges (since a shortest path in a graph without negative cycles never repeats a vertex).

We prove by induction on i that after pass i, `distTo[v_i] = delta(s, v_i)`.

Base case (i = 0): `distTo[s] = 0 = delta(s, s)`, established before any passes.

Inductive step: Assume after pass i - 1, `distTo[v_{i-1}] = delta(s, v_{i-1})`. In pass i, we relax the edge (v_{i-1}, v_i). After relaxation:

```
distTo[v_i] <= distTo[v_{i-1}] + w(v_{i-1}, v_i)
            = delta(s, v_{i-1}) + w(v_{i-1}, v_i)
            = delta(s, v_i)
```

Since `distTo[v_i] >= delta(s, v_i)` always (relaxation never produces an underestimate), we conclude `distTo[v_i] = delta(s, v_i)`.

After V - 1 passes, this holds for all vertices on any shortest path of length at most V - 1, which includes all vertices reachable from s. QED.

6.4 Negative Cycle Detection

DEFINITION (Negative Cycle). A *negative cycle* is a directed cycle whose total edge weight is negative. If a negative cycle is reachable from the source s, then shortest paths to vertices reachable from the cycle are undefined (we can make the path arbitrarily short by going around the cycle more times).

PROPOSITION K. If a V-th pass of Bellman-Ford successfully relaxes any edge, then G contains a negative cycle reachable from s.

Proof. After V - 1 passes, if no negative cycle exists, then `distTo[v] = delta(s, v)` for all reachable v (Proposition J), so no further relaxation is possible. Conversely, if a V-th relaxation succeeds, then the distTo values have not converged, which implies the existence of a negative cycle.

To find the actual cycle: if edge (v, w) is relaxed in the V-th pass, trace back through `edgeTo[]` from v. Since the path must eventually revisit a vertex (by the pigeonhole principle), the repeated vertex identifies a cycle. Collect all edges on this cycle; their total weight is negative. QED.

6.5 Detailed Trace

Consider a small directed graph with 5 vertices (0 through 4), source = 0:

```
Directed edges:
  0 -> 1 (6)     0 -> 3 (7)
  1 -> 2 (5)     1 -> 3 (8)     1 -> 4 (-4)
  2 -> 1 (-2)
  3 -> 2 (-3)    3 -> 4 (9)
  4 -> 0 (2)     4 -> 2 (7)
```

Edge order for relaxation: 0->1, 0->3, 1->2, 1->3, 1->4, 2->1, 3->2, 3->4, 4->0, 4->2.

Initial: distTo = [0, inf, inf, inf, inf]

Pass 1:

- Relax 0->1: distTo[1] = 0 + 6 = 6
- Relax 0->3: distTo[3] = 0 + 7 = 7
- Relax 1->2: distTo[2] = 6 + 5 = 11
- Relax 1->3: 6 + 8 = 14 > 7. No update.
- Relax 1->4: distTo[4] = 6 + (-4) = 2
- Relax 2->1: 11 + (-2) = 9 > 6. No update.
- Relax 3->2: 7 + (-3) = 4 < 11. distTo[2] = 4
- Relax 3->4: 7 + 9 = 16 > 2. No update.
- Relax 4->0: 2 + 2 = 4 > 0. No update.
- Relax 4->2: 2 + 7 = 9 > 4. No update.
- After pass 1: distTo = [0, 6, 4, 7, 2]

Pass 2:

- Relax 0->1: 0 + 6 = 6. No update.
- Relax 0->3: 0 + 7 = 7. No update.
- Relax 1->2: 6 + 5 = 11 > 4. No update.
- Relax 1->3: 6 + 8 = 14 > 7. No update.
- Relax 1->4: 6 + (-4) = 2. No update.
- Relax 2->1: 4 + (-2) = 2 < 6. distTo[1] = 2
- Relax 3->2: 7 + (-3) = 4. No update.
- Relax 3->4: 7 + 9 = 16 > 2. No update.
- Relax 4->0: 2 + 2 = 4 > 0. No update.
- Relax 4->2: 2 + 7 = 9 > 4. No update.
- After pass 2: distTo = [0, 2, 4, 7, 2]

Pass 3:

- Relax 0->1: 0 + 6 = 6 > 2. No update.
- Relax 0->3: 0 + 7 = 7. No update.
- Relax 1->2: 2 + 5 = 7 > 4. No update.
- Relax 1->3: 2 + 8 = 10 > 7. No update.
- Relax 1->4: 2 + (-4) = -2 < 2. distTo[4] = -2
- Relax 2->1: 4 + (-2) = 2. No update.
- Relax 3->2: 7 + (-3) = 4. No update.
- Relax 3->4: 7 + 9 = 16 > -2. No update.
- Relax 4->0: -2 + 2 = 0. No update.
- Relax 4->2: -2 + 7 = 5 > 4. No update.
- After pass 3: distTo = [0, 2, 4, 7, -2]

Pass 4 (final, since V - 1 = 4):

- Relax all edges. No updates occur (all distances have converged).
- After pass 4: distTo = [0, 2, 4, 7, -2]

Negative cycle check (pass 5):

- Relax all edges. No edge produces a shorter path. No negative cycle detected.

Final shortest-path distances from vertex 0:

```
Vertex:   0    1    2    3    4
distTo:   0    2    4    7   -2
```

Shortest paths:

```
0 -> 1: 0 -> 3 -> 2 -> 1   (weight: 7 + (-3) + (-2) = 2)
0 -> 2: 0 -> 3 -> 2         (weight: 7 + (-3) = 4)
0 -> 3: 0 -> 3              (weight: 7)
0 -> 4: 0 -> 3 -> 2 -> 1 -> 4 (weight: 7 + (-3) + (-2) + (-4) = -2)
```

6.6 Time Complexity Analysis

PROPOSITION L. The Bellman-Ford algorithm runs in O(VE) time and O(V) extra space.

Proof. The algorithm performs V - 1 passes, each of which relaxes all E edges. Each relaxation takes O(1) time. The negative-cycle detection performs one additional pass over all E edges. Total time: O(VE). The algorithm uses O(V) space for `distTo` and `edgeTo` arrays, plus the input graph. QED.

6.7 Optimizations

Early termination. If a pass produces no relaxations, the algorithm can terminate early since all distances have converged. In the best case (when the source has a short path to every vertex), this can reduce the number of passes significantly.

Queue-based Bellman-Ford (SPFA). The Shortest Path Faster Algorithm maintains a queue of vertices whose `distTo` values changed in the previous pass. Only edges from these vertices need to be relaxed. In practice, SPFA is often much faster than the basic version, but its worst-case complexity remains O(VE).

```
function SPFA(G, s):
    distTo = array[0..V-1] of real, all +infinity
    edgeTo = array[0..V-1] of DirectedEdge, all null
    onQueue = array[0..V-1] of boolean, all false
    distTo[s] = 0.0

    queue = new Queue
    queue.enqueue(s)
    onQueue[s] = true

    while queue is not empty:
        v = queue.dequeue()
        onQueue[v] = false

        for each directed edge e = (v, w, weight) from v:
            if distTo[v] + weight < distTo[w]:
                distTo[w] = distTo[v] + weight
                edgeTo[w] = e
                if not onQueue[w]:
                    queue.enqueue(w)
                    onQueue[w] = true

    return (distTo, edgeTo)
```

Note: negative cycle detection in SPFA typically involves counting the number of times each vertex enters the queue and flagging a cycle if any vertex enters more than V times.

6.8 Negative Cycles in Practice

Negative cycles are not merely a theoretical curiosity. In currency exchange, each edge represents an exchange rate. If the product of exchange rates around a cycle exceeds 1.0, the cycle represents an arbitrage opportunity. Taking logarithms converts multiplication to addition and rates > 1 to negative weights, so detecting arbitrage reduces to detecting negative cycles.

Similarly, in project scheduling with penalties, negative cycles indicate infeasible constraints (circular dependencies with conflicting deadlines).

Part VII: Algorithm Comparison

7.1 Summary Table

The following table compares the four algorithms studied in this lecture:

```
+------------------+-------------+---------+----------+-----------+---------------+
| Algorithm        | Problem     | Weights | Time     | Space     | Data Struct.  |
+------------------+-------------+---------+----------+-----------+---------------+
| Kruskal's        | MST         | Any     | O(E lg E)| O(V + E)  | Union-Find    |
| Prim's (eager)   | MST         | Any     | O(E lg V)| O(V)      | Indexed PQ    |
| Dijkstra's       | SSSP        | >= 0    | O(E lg V)| O(V)      | Indexed PQ    |
| Bellman-Ford     | SSSP        | Any     | O(VE)    | O(V)      | None (arrays) |
+------------------+-------------+---------+----------+-----------+---------------+
```

Key:
- MST = Minimum Spanning Tree (undirected graphs)
- SSSP = Single-Source Shortest Paths (directed graphs)
- lg = log base 2

7.2 Decision Guide

When choosing an algorithm:

1. **MST problem?** Use Kruskal's for sparse graphs or when edges are pre-sorted; use Prim's (eager) for dense graphs or when starting from a specific vertex.
2. **Shortest paths with non-negative weights?** Use Dijkstra's. It is the fastest practical algorithm for this case.
3. **Shortest paths with negative weights?** Use Bellman-Ford. If you also need negative-cycle detection, use the V-th pass check.
4. **Shortest paths in a DAG?** Use topological sort + relaxation in O(V + E), regardless of edge-weight signs.
5. **All-pairs shortest paths?** Run Dijkstra from each vertex (O(VE log V)) for non-negative weights, or use the Floyd-Warshall algorithm (O(V^3)) for general weights.

7.3 Structural Parallels

Prim's and Dijkstra's algorithms are structurally nearly identical. Both maintain a set of "settled" vertices and grow it by extracting the minimum-priority vertex from an indexed priority queue. The key difference is the priority value:

- **Prim's:** priority of v = weight of the minimum-weight edge from v to the tree.
- **Dijkstra's:** priority of v = total distance from source to v.

This small difference leads to fundamentally different results: Prim's minimizes total edge weight (MST), while Dijkstra's minimizes path distance from the source (SPT).

Similarly, Kruskal's and Bellman-Ford share a structural theme of processing *all* edges repeatedly, though Kruskal's processes them once in sorted order while Bellman-Ford processes them V - 1 times in arbitrary order.

Part VIII: Practice Problems

Problem 1: Kruskal's MST Trace

Problem. Apply Kruskal's algorithm to the following graph and show the state of the Union-Find structure after each edge addition.

```
Vertices: A, B, C, D, E, F
Edges:
  (A,B,4)  (A,F,2)  (B,C,6)  (B,F,5)
  (C,D,3)  (C,F,1)  (D,E,2)  (E,F,4)
```

Solution.

Step 0: Sort edges by weight:

```
(C,F,1), (A,F,2), (D,E,2), (C,D,3), (A,B,4), (E,F,4), (B,F,5), (B,C,6)
```

Initial UF: {A}, {B}, {C}, {D}, {E}, {F}. Components = 6.

Step 1: Process (C,F,1). find(C) != find(F). Accept. Union(C,F).
- UF: {A}, {B}, {C,F}, {D}, {E}. Components = 5. MST edges: 1.

Step 2: Process (A,F,2). find(A) != find(F). Accept. Union(A,F).
- UF: {A,C,F}, {B}, {D}, {E}. Components = 4. MST edges: 2.

Step 3: Process (D,E,2). find(D) != find(E). Accept. Union(D,E).
- UF: {A,C,F}, {B}, {D,E}. Components = 3. MST edges: 3.

Step 4: Process (C,D,3). find(C) = root of {A,C,F}. find(D) = root of {D,E}. Different. Accept. Union.
- UF: {A,C,D,E,F}, {B}. Components = 2. MST edges: 4.

Step 5: Process (A,B,4). find(A) = root of {A,C,D,E,F}. find(B) = B. Different. Accept. Union.
- UF: {A,B,C,D,E,F}. Components = 1. MST edges: 5 = V - 1. Done.

MST edges: (C,F,1), (A,F,2), (D,E,2), (C,D,3), (A,B,4). **Total weight = 12.**

Edges (E,F,4), (B,F,5), (B,C,6) were not considered (algorithm terminated early) or would have been rejected.

Problem 2: Prim's MST Trace

Problem. Apply Prim's algorithm (eager version) to the same graph as Problem 1, starting from vertex A. Show the state of the indexed priority queue at each step.

Solution.

Initial: distTo = [A:0, B:inf, C:inf, D:inf, E:inf, F:inf]. IPQ = {A:0}.

Step 1: Extract A. Mark A. Process edges from A:
- A-B (4): distTo[B] = 4, edgeTo[B] = A-B. Insert B.
- A-F (2): distTo[F] = 2, edgeTo[F] = A-F. Insert F.
- IPQ = {F:2, B:4}

Step 2: Extract F. Mark F. MST edge: A-F (2). Process edges from F:
- A-F: A marked. Skip.
- B-F (5): 5 > 4. No update.
- C-F (1): distTo[C] = 1, edgeTo[C] = C-F. Insert C.
- E-F (4): distTo[E] = 4, edgeTo[E] = E-F. Insert E.
- IPQ = {C:1, B:4, E:4}

Step 3: Extract C. Mark C. MST edge: C-F (1). Process edges from C:
- C-D (3): distTo[D] = 3, edgeTo[D] = C-D. Insert D.
- C-F: F marked. Skip.
- B-C (6): 6 > 4. No update.
- IPQ = {D:3, B:4, E:4}

Step 4: Extract D. Mark D. MST edge: C-D (3). Process edges from D:
- C-D: C marked. Skip.
- D-E (2): 2 < 4. distTo[E] = 2, edgeTo[E] = D-E. DecreaseKey E.
- IPQ = {E:2, B:4}

Step 5: Extract E. Mark E. MST edge: D-E (2). Process edges from E:
- D-E: D marked. Skip.

- E-F: F marked. Skip.
- IPQ = {B:4}

Step 6: Extract B. Mark B. MST edge: A-B (4). Process edges from B:
- All neighbors marked. IPQ empty.

MST edges: A-F (2), C-F (1), C-D (3), D-E (2), A-B (4). **Total weight = 12.** Same MST, different order.

Problem 3: Dijkstra's Shortest Path Trace

Problem. Apply Dijkstra's algorithm to the following directed graph with source vertex S. Show the extraction order and reconstruct shortest paths.

```
Vertices: S, A, B, C, D, T
Directed edges:
  S->A (10)   S->C (5)
  A->B (1)    A->C (2)
  B->T (4)
  C->A (3)    C->B (9)    C->D (2)
  D->S (7)    D->T (6)
```

Solution.

Initial: distTo = [S:0, A:inf, B:inf, C:inf, D:inf, T:inf]. IPQ = {S:0}.

Step 1: Extract S (dist 0). Mark S.
- S->A: distTo[A] = 10. Insert A.
- S->C: distTo[C] = 5. Insert C.
- IPQ = {C:5, A:10}

Step 2: Extract C (dist 5). Mark C.
- C->A: 5+3=8 < 10. distTo[A] = 8. DecreaseKey A.
- C->B: 5+9=14. distTo[B] = 14. Insert B.
- C->D: 5+2=7. distTo[D] = 7. Insert D.
- IPQ = {D:7, A:8, B:14}

Step 3: Extract D (dist 7). Mark D.
- D->S: S marked. Skip.
- D->T: 7+6=13. distTo[T] = 13. Insert T.
- IPQ = {A:8, T:13, B:14}

Step 4: Extract A (dist 8). Mark A.
- A->B: 8+1=9 < 14. distTo[B] = 9. DecreaseKey B.
- A->C: C marked. Skip.
- IPQ = {B:9, T:13}

Step 5: Extract B (dist 9). Mark B.
- B->T: 9+4=13. 13 = 13. No update (not strictly less).
- IPQ = {T:13}

Step 6: Extract T (dist 13). Mark T. No outgoing edges in our graph from T. Done.

Final distances: S:0, A:8, B:9, C:5, D:7, T:13.

Shortest paths (via edgeTo reconstruction):
- S->A: S->C->A (5+3 = 8)
- S->B: S->C->A->B (5+3+1 = 9)
- S->C: S->C (5)
- S->D: S->C->D (5+2 = 7)
- S->T: S->C->D->T (5+2+6 = 13) or S->C->A->B->T (5+3+1+4 = 13). Both have length 13; the algorithm found one of them (depending on tie-breaking and edge processing order).

Problem 4: Prove the Cut Property

Problem. Let G be a connected, edge-weighted, undirected graph where all edge weights are distinct. Prove that G has a unique MST.

Solution.

Proof. Suppose for contradiction that G has two distinct MSTs, T1 and T2. Since T1 != T2, there exists an edge e that is in T1 but not in T2.

Adding e to T2 creates a unique cycle C in T2 + {e} (since T2 is a spanning tree). Some edge f on this cycle is in T2 but not in T1 (otherwise T1 would contain the entire cycle, contradicting T1 being acyclic).

Since all edge weights are distinct, either w(e) < w(f) or w(e) > w(f).

Case 1: w(e) < w(f). Consider T2' = T2 - {f} + {e}. Removing f from T2 disconnects it into two components; adding e (which is on the cycle created by f's removal) reconnects them. So T2' is a spanning tree with w(T2') = w(T2) - w(f) + w(e) < w(T2). This contradicts T2 being an MST.

Case 2: w(e) > w(f). By a symmetric argument, T1' = T1 - {e} + {f} is a spanning tree with w(T1') < w(T1), contradicting T1 being an MST.

Both cases lead to contradiction, so T1 = T2. The MST is unique. QED.

Problem 5: Union-Find Operations Trace

Problem. Starting from 10 elements {0, 1, 2, ..., 9}, perform the following union operations using union by rank with path compression (halving). Show the parent and rank arrays after each operation.

Operations: union(3,4), union(4,9), union(8,0), union(2,3), union(5,6), union(5,9), union(7,3), union(4,8), union(6,1)

Solution.

Initial:

```
Index:  0  1  2  3  4  5  6  7  8  9
Parent: 0  1  2  3  4  5  6  7  8  9
Rank:   0  0  0  0  0  0  0  0  0  0
```

union(3,4): find(3)=3, find(4)=4. Equal rank. Parent[4]=3, Rank[3]=1.

```
Parent: 0  1  2  3  3  5  6  7  8  9
Rank:   0  0  0  1  0  0  0  0  0  0
```

union(4,9): find(4): 4->3, return 3. find(9)=9. Rank[3]=1 > Rank[9]=0. Parent[9]=3.

```
Parent: 0  1  2  3  3  5  6  7  8  3
Rank:   0  0  0  1  0  0  0  0  0  0
```

union(8,0): find(8)=8, find(0)=0. Equal rank. Parent[0]=8, Rank[8]=1.

```
Parent: 8  1  2  3  3  5  6  7  8  3
Rank:   0  0  0  1  0  0  0  0  1  0
```

union(2,3): find(2)=2, find(3)=3. Rank[3]=1 > Rank[2]=0. Parent[2]=3.

```
Parent: 8  1  3  3  3  5  6  7  8  3
Rank:   0  0  0  1  0  0  0  0  1  0
```

union(5,6): find(5)=5, find(6)=6. Equal rank. Parent[6]=5, Rank[5]=1.

```
Parent: 8  1  3  3  3  5  5  7  8  3
Rank:   0  0  0  1  0  1  0  0  1  0
```

union(5,9): find(5)=5. find(9): 9->3, return 3. Rank[3]=1 = Rank[5]=1. Parent[5]=3, Rank[3]=2.

```
Parent: 8  1  3  3  3  3  5  7  8  3
Rank:   0  0  0  2  0  1  0  0  1  0
```

union(7,3): find(7)=7. find(3)=3. Rank[3]=2 > Rank[7]=0. Parent[7]=3.

```
Parent: 8  1  3  3  3  3  5  3  8  3
Rank:   0  0  0  2  0  1  0  0  1  0
```

union(4,8): find(4): 4->3, return 3. find(8)=8. Rank[3]=2 > Rank[8]=1. Parent[8]=3.

```
Parent: 8  1  3  3  3  3  5  3  3  3
Rank:   0  0  0  2  0  1  0  0  1  0
```

Note: vertex 0's parent is 8, and 8's parent is now 3. So find(0) would follow 0->8->3 with path halving: set parent[0]=parent[8]=3, then follow 8->3. After path compression, parent[0]=3.

union(6,1): find(6): 6->5->3, with path halving: parent[6]=parent[5]=3. Return 3. find(1)=1. Rank[3]=2 > Rank[1]=0. Parent[1]=3.

```
Parent: 8  3  3  3  3  3  3  3  3  3
Rank:   0  0  0  2  0  1  0  0  1  0
```

After all operations, all elements are in one component rooted at 3.

Problem 6: Shortest Path with Path Reconstruction

Problem. Run Dijkstra's algorithm on the following graph from source 0. Reconstruct and print the shortest path from 0 to 5.

```
Vertices: 0, 1, 2, 3, 4, 5
Directed edges:
  0->1 (2)    0->2 (4)
  1->2 (1)    1->3 (7)
  2->4 (3)
  3->5 (1)
  4->3 (2)    4->5 (5)
```

Solution.

Initial: distTo = [0, inf, inf, inf, inf, inf]. IPQ = {0:0}.

Extract 0: Relax 0->1 (distTo[1]=2), 0->2 (distTo[2]=4). IPQ = {1:2, 2:4}.

Extract 1: Relax 1->2 (2+1=3 < 4, distTo[2]=3, edgeTo[2]=1->2), 1->3 (2+7=9, distTo[3]=9). IPQ = {2:3, 3:9}.

Extract 2: Relax 2->4 (3+3=6, distTo[4]=6). IPQ = {4:6, 3:9}.

Extract 4: Relax 4->3 (6+2=8 < 9, distTo[3]=8, edgeTo[3]=4->3), 4->5 (6+5=11, distTo[5]=11). IPQ = {3:8, 5:11}.

Extract 3: Relax 3->5 (8+1=9 < 11, distTo[5]=9, edgeTo[5]=3->5). IPQ = {5:9}.

Extract 5: No outgoing edges. Done.

Final distances: distTo = [0, 2, 3, 8, 6, 9].

Path reconstruction for vertex 5:
- edgeTo[5] = 3->5. Previous vertex: 3.
- edgeTo[3] = 4->3. Previous vertex: 4.
- edgeTo[4] = 2->4. Previous vertex: 2.
- edgeTo[2] = 1->2. Previous vertex: 1.
- edgeTo[1] = 0->1. Previous vertex: 0 (source).

Shortest path from 0 to 5: 0 -> 1 -> 2 -> 4 -> 3 -> 5, total weight = 2 + 1 + 3 + 2 + 1 = **9**.

Problem 7: Bellman-Ford with Negative Cycle

Problem. Apply Bellman-Ford to the following graph from source A. Determine whether a negative cycle exists.

```
Vertices: A, B, C, D
Directed edges:
  A->B (1)
  B->C (3)    B->D (2)
  C->D (5)    C->B (-7)
  D->A (4)
```

Solution.

Edge order: A->B, B->C, B->D, C->D, C->B, D->A.

Initial: distTo = [A:0, B:inf, C:inf, D:inf].

Pass 1:
- A->B: distTo[B] = 0+1 = 1.
- B->C: distTo[C] = 1+3 = 4.
- B->D: distTo[D] = 1+2 = 3.
- C->D: 4+5=9 > 3. No update.
- C->B: 4+(-7) = -3 < 1. distTo[B] = -3.

- D->A: 3+4=7 > 0. No update.
- After pass 1: [A:0, B:-3, C:4, D:3]

Pass 2:
- A->B: 0+1=1 > -3. No update.
- B->C: -3+3=0 < 4. distTo[C] = 0.
- B->D: -3+2=-1 < 3. distTo[D] = -1.
- C->D: 0+5=5 > -1. No update.
- C->B: 0+(-7)=-7 < -3. distTo[B] = -7.
- D->A: -1+4=3 > 0. No update.
- After pass 2: [A:0, B:-7, C:0, D:-1]

Pass 3 (V - 1 = 3):
- A->B: 1 > -7. No update.
- B->C: -7+3=-4 < 0. distTo[C] = -4.
- B->D: -7+2=-5 < -1. distTo[D] = -5.
- C->D: -4+5=1 > -5. No update.
- C->B: -4+(-7)=-11 < -7. distTo[B] = -11.
- D->A: -5+4=-1 < 0. distTo[A] = -1.
- After pass 3: [A:-1, B:-11, C:-4, D:-5]

Distances are still changing! This is a strong indication of a negative cycle.

Negative cycle check (pass 4):
- A->B: -1+1=0 > -11. No update.
- B->C: -11+3=-8 < -4. **Relaxation occurred!**

Negative cycle detected.

The cycle B -> C -> B has weight 3 + (-7) = -4 < 0. Every traversal of this cycle reduces the path weight by 4, so distances can be made arbitrarily negative.

Problem 8: MST Edge Weight Analysis

Problem. Prove that the maximum-weight edge in any MST of a connected graph G is the minimum possible over all spanning trees. That is, if e_max(T) denotes the maximum edge weight in spanning tree T, then for the MST T, *e_max(T)* <= e_max(T) for all spanning trees T.

Solution.

Proof. Let T *be an MST and let e* be the maximum-weight edge in T. *Suppose for contradiction that some spanning tree T has e_max(T) < w(e*).

Removing e *from T* partitions V into two components S and V - S. Since T is a spanning tree, T must contain some edge f that crosses this same cut (S, V - S). We know w(f) <= e_max(T) < w(e*) (since f is in T and e_max(T) < w(e*)* by assumption).

Consider T' = T - *{e}* + {f}. Since e* crosses the cut (S, V - S) and f also crosses this cut, T' is still a spanning tree. Its total weight is:

```
w(T') = w(T*) - w(e*) + w(f) < w(T*)
```

This contradicts T *being a minimum spanning tree. Therefore no spanning tree T has e_max(T) < e_max(T*). QED.

This result is the foundation of the **minimax path** problem: the maximum edge weight on the path between two vertices in the MST is the minimum possible over all paths between those vertices in the original graph.

Summary

This lecture covered four fundamental algorithms for optimization on weighted graphs.

Minimum Spanning Trees are the cheapest way to connect all vertices of an undirected weighted graph. The cut property provides the theoretical foundation: the minimum-weight edge crossing any cut must be in the MST. Two classic algorithms exploit this property:

- **Kruskal's algorithm** processes edges globally in weight order, using Union-Find (with path compression and union by rank) to detect cycles. Its O(E log E) running time is dominated by sorting.

- **Prim's algorithm** grows the MST from a source vertex, using an indexed priority queue to efficiently select the cheapest crossing edge. The eager version runs in O(E log V) time with O(V) extra space.

Shortest Paths find the minimum-weight path from a source to all other vertices in a directed weighted graph. The relaxation operation is the fundamental building block:

- **Dijkstra's algorithm** is the method of choice for non-negative edge weights. It processes vertices in order of distance from the source, using an indexed priority queue. Its O(E log V) running time and correctness proof rely crucially on the non-negativity assumption.

- **Bellman-Ford** handles arbitrary edge weights (including negative) by relaxing all edges V - 1 times. Its O(VE) running time is slower than Dijkstra's, but it can detect negative cycles, which is critical for applications like arbitrage detection.

Key theoretical results proved in this lecture:
- Cut property (Proposition A) and cycle property (Proposition B)
- Uniqueness of MST when edge weights are distinct (Problem 4)
- Correctness of Dijkstra's algorithm by contradiction (Proposition H)
- Sufficiency of V - 1 passes in Bellman-Ford (Proposition J)
- Union-Find amortized O(alpha(N)) per operation (Proposition D, proof sketched)

Algorithm comparison (final summary):

```
+------------------+---------+-----------+----------+------------------+
| Algorithm        | Problem | Time      | Neg. Wt? | Key Structure    |
+------------------+---------+-----------+----------+------------------+
| Kruskal's        | MST     | O(E lg E) | N/A      | Union-Find       |
| Prim's (eager)   | MST     | O(E lg V) | N/A      | Indexed Min-PQ   |
| Dijkstra's       | SSSP    | O(E lg V) | No       | Indexed Min-PQ   |
| Bellman-Ford     | SSSP    | O(VE)     | Yes      | None (brute)     |
+------------------+---------+-----------+----------+------------------+
```

The companion C++ code implements all four algorithms: `Edge` struct, `WeightedGraph`, `UnionFind` (path compression + union by rank), `kruskalMST`, `primMST` (eager with min-heap), and `dijkstra` with full path reconstruction.

References

1. Sedgewick, R. & Wayne, K. (2011). *Algorithms*, 4th Edition. Addison-Wesley. Chapters 4.3 (Minimum Spanning Trees) and 4.4 (Shortest Paths).

2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapters 21 (Disjoint Sets), 23 (Minimum Spanning Trees), 24 (Single-Source Shortest Paths).

3. Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs." *Numerische Mathematik*, 1(1), 269--271.

4. Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem." *Proceedings of the American Mathematical Society*, 7(1), 48--50.

5. Prim, R. C. (1957). "Shortest connection networks and some generalizations." *Bell System Technical Journal*, 36(6), 1389--1401.

6. Bellman, R. (1958). "On a routing problem." *Quarterly of Applied Mathematics*, 16(1), 87--90.

7. Tarjan, R. E. (1975). "Efficiency of a good but not linear set union algorithm." *Journal of the ACM*, 22(2), 215--225.

8. Fredman, M. L. & Tarjan, R. E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms." *Journal of the ACM*, 34(3), 596--615.

9. Jarnik, V. (1930). "O jistem problemu minimalnim." *Prace Moravske Prirodovedecke Spolecnosti*, 6, 57--63.

10. Seidel, R. & Sharir, M. (2005). "Top-down analysis of path compression." *SIAM Journal on Computing*, 34(3), 515--525.

Lecture 11 of 12 -- Data Structures & Algorithms -- Based on Sedgewick's Algorithms, 4th Edition