Lab Session 8

Minimum Spanning Trees & Shortest Paths

The Art of Computer Programming 2 — Week 8 — April 15, 2026

Objectives

By the end of this lab, you should be able to:

- Trace Kruskal's and Prim's algorithms by hand on a small graph
- Implement Union-Find with path compression and union by rank
- Implement Kruskal's and Prim's MST algorithms in Python
- Trace Bellman-Ford and Dijkstra step by step
- Implement Bellman-Ford and Dijkstra in Python
- Understand when each algorithm applies (negative weights, DAGs, etc.)

1 Warm-up: MST by Hand (15 min)

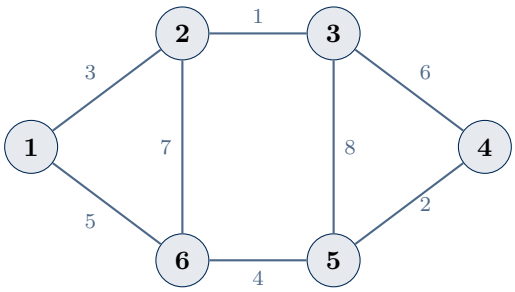
Pen and paper — no computer needed.

1.1 Kruskal's algorithm (CLRS 21.2)

Recall

Kruskal's algorithm: sort all edges by nondecreasing weight. Process edges one by one: add the edge to the MST if it connects two different components (using Union-Find). Skip it if both endpoints are already in the same component. Repeat until $|V| - 1$ edges have been added.

Consider the following weighted undirected graph with 6 vertices:



(a) List all edges sorted by weight.

(b) Run Kruskal's algorithm step by step. For each edge, indicate whether it is **added** or **skipped** (cycle), and show the components after each step.

(c) What is the total weight of the MST?

1.2 Prim's algorithm (CLRS 21.2)

Recall

Prim's algorithm: grow a tree from a root vertex. At each step, add the lightest edge connecting the tree to a non-tree vertex. Use a min-priority queue keyed by the minimum edge weight from each non-tree vertex to the tree.

Using the **same graph** above:

(a) Run Prim's algorithm starting from vertex 1. Show the priority queue contents after each extraction.

(b) Do Kruskal and Prim produce the same MST? Why or why not?

Hint

For Prim: after extracting a vertex u, update the keys of all neighbors v not yet in the tree. The key of v is the minimum weight of an edge connecting v to any tree vertex.

2 Implement Union-Find (10 min)

Recall

Union-Find (Disjoint-Set Union) supports:

- MAKE-SET(v): create a singleton set
- FIND(v): return the representative of v's set (with **path compression**)
- UNION(u, v): merge the sets of u and v (with **union by rank**)

With both optimizations, m operations on n elements take $\mathcal{O}(m \cdot \alpha(n))$ time.

```python
class UnionFind:
    def __init__(self, n):
        """Create n singleton sets (vertices 0..n-1)."""
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        """Find with path compression."""
        # YOUR CODE HERE
        pass

    def union(self, x, y):
        """Union by rank. Returns True if x,y were in different sets."""
        # YOUR CODE HERE
        pass
```

```python
# Test Union-Find
uf = UnionFind(6)
assert uf.find(0) != uf.find(1)
assert uf.union(0, 1) == True
assert uf.find(0) == uf.find(1)
assert uf.union(0, 1) == False  # already in same set
uf.union(2, 3)
uf.union(0, 2)
assert uf.find(0) == uf.find(3)  # transitive
assert uf.find(0) != uf.find(4)
print("Union-Find tests passed!")
```

3 Implement Kruskal's Algorithm (15 min)

```python
def kruskal(n, edges):
    """
    Compute the MST using Kruskal's algorithm.
    Args:
        n: number of vertices (labeled 0..n-1)
        edges: list of (weight, u, v)
    Returns:
        list of (u, v, weight) in the MST
    """
    # YOUR CODE HERE
    # 1. Sort edges by weight
    # 2. Initialize Union-Find
    # 3. For each edge in sorted order:
    #    if endpoints are in different components: add to MST and union
    # 4. Stop when MST has n-1 edges
    pass
```

```python
# Test on the graph from the warm-up
#   1--2 (3), 2--3 (1), 3--4 (6), 4--5 (2), 5--6 (4), 6--1 (5),
#   2--6 (7), 3--5 (8)
# Vertices: 0=1, 1=2, 2=3, 3=4, 4=5, 5=6
edges = [
    (3, 0, 1), (1, 1, 2), (6, 2, 3), (2, 3, 4),
    (4, 4, 5), (5, 5, 0), (7, 1, 5), (8, 2, 4)
]

mst = kruskal(6, edges)
total_weight = sum(w for _, _, w in mst)
print(f"Kruskal MST: {mst}")
print(f"Total weight: {total_weight}")

assert len(mst) == 5  # n-1 edges
assert total_weight == 15
print("Kruskal test passed!")
```

4 Implement Prim's Algorithm (15 min)

Recall

In Python, use `heapq` as a min-priority queue with the "lazy deletion" pattern: push `(key, vertex)` entries, and skip stale entries when popping (if the vertex is already in the tree).

```python
import heapq

def prim(graph, r=0):
    """
    Compute the MST using Prim's algorithm.
    Args:
        graph: adjacency list, graph[u] = [(v, weight), ...]
        r: root vertex
    Returns:
        list of (u, v, weight) in the MST
    """
```

```
    # YOUR CODE HERE
    # 1. Initialize key[v] = inf, parent[v] = None, in_tree[v] = False
    # 2. key[r] = 0, push (0, r) into priority queue
    # 3. While PQ is not empty:
    #    - Pop (k, u). If in_tree[u], skip (stale entry).
    #    - Mark in_tree[u] = True
    #    - If parent[u] is not None, add (parent[u], u, k) to MST
    #    - For each neighbor (v, w): if not in_tree[v] and w < key[v]:
    #        update key[v], parent[v], push (w, v)
    pass
```

```
# Build adjacency list for the same graph
graph = [[] for _ in range(6)]
edge_list = [(0,1,3), (1,2,1), (2,3,6), (3,4,2), (4,5,4), (5,0,5), (1,5,7), (2,4,8)]
for u, v, w in edge_list:
    graph[u].append((v, w))
    graph[v].append((u, w))

mst = prim(graph, r=0)
total_weight = sum(w for _, _, w in mst)
print(f"Prim MST: {mst}")
print(f"Total weight: {total_weight}")

assert len(mst) == 5
assert total_weight == 15
print("Prim test passed!")
```

5 Warm-up: Shortest Paths by Hand (15 min)

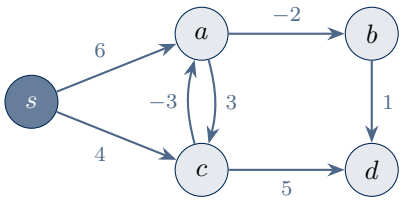
Pen and paper — no computer needed.

5.1 Bellman-Ford (CLRS 22.1)

Recall

Bellman-Ford: handles negative-weight edges. Performs $|V| - 1$ passes over all edges, relaxing each one. After pass i, all shortest paths with $\leqslant i$ edges are correct. A final pass detects negative-weight cycles.
Relax(u, v, w): if $v.d > u.d + w(u, v)$, set $v.d = u.d + w(u, v)$ and $v.\pi = u$.

Run Bellman-Ford on the following directed graph with source s:



(a) Fill the distance estimates $d[v]$ after each pass (initial, pass 1, pass 2, pass 3, pass 4).

(b) Does the graph contain a negative-weight cycle reachable from s?

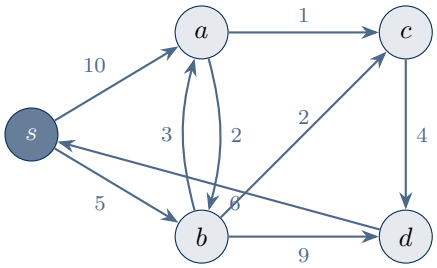
(c) Give the shortest-path tree (the π values).

5.2 Dijkstra (CLRS 22.3)

Recall

Dijkstra: for graphs with nonnegative weights only. Maintains a set S of finalized vertices. At every step, extract the vertex u with minimum d-value from the priority queue, add it to S, and relax all its outgoing edges.

Run Dijkstra on the following directed graph with source s:



(a) Show the extraction order and the d-values at each step.

(b) Give the shortest-path tree (i.e., the π values).

(c) What is the shortest path from s to d, and its weight?

6 Implement Bellman-Ford (15 min)

```python
def bellman_ford(n, edges, s):
    """
    Single-source shortest paths via Bellman-Ford.
    Args:
        n: number of vertices (0..n-1)
        edges: list of (u, v, weight)
        s: source vertex
    Returns:
        (dist, parent, has_negative_cycle)
    """
    # YOUR CODE HERE
    # 1. Initialize dist[v] = inf, parent[v] = None; dist[s] = 0
    # 2. Repeat n-1 times: for each edge (u,v,w), relax
    # 3. Check for negative cycle: one more pass over all edges
    pass
```

```python
def reconstruct_path(parent, s, v):
    """Reconstruct shortest path from s to v using parent pointers."""
    path = []
    current = v
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    if path[0] != s:
        return []  # no path
    return path
```

```python
# Test on Bellman-Ford graph from warm-up
# s=0, a=1, b=2, c=3, d=4
edges_bf = [
    (0, 1, 6),   # s -> a : 6
    (0, 3, 4),   # s -> c : 4
    (1, 2, -2),  # a -> b : -2
    (1, 3, 3),   # a -> c : 3
    (3, 4, 5),   # c -> d : 5
    (2, 4, 1),   # b -> d : 1
    (3, 1, -3),  # c -> a : -3
]

dist, parent, neg_cycle = bellman_ford(5, edges_bf, 0)
print(f"Distances: {dist}")
print(f"Parents:   {parent}")
print(f"Negative cycle: {neg_cycle}")

# Expected distances: s=0, a=1, b=-1, c=4, d=0
assert dist == [0, 1, -1, 4, 0]
assert neg_cycle == False

path_to_d = reconstruct_path(parent, 0, 4)
print(f"Path s->d: {path_to_d}")
print("Bellman-Ford test passed!")
```

Important

Bellman-Ford runs in $\mathcal{O}(VE)$ — much slower than Dijkstra for nonneg. weights. Use it only when negative edges are possible.

7 Implement Dijkstra's Algorithm (15 min)

Recall

Use the "lazy deletion" pattern with `heapq`: push `(dist, vertex)` and skip entries where `d > dist[u]` when popping.

```python
import heapq

def dijkstra(graph, s):
    """
    Single-source shortest paths via Dijkstra (nonneg. weights).
    Args:
        graph: adjacency list, graph[u] = [(v, weight), ...]
        s: source vertex
    Returns:
        (dist, parent)
    """
    # YOUR CODE HERE
    # 1. Initialize dist[v] = inf, parent[v] = None; dist[s] = 0
    # 2. Push (0, s) into priority queue
    # 3. While PQ not empty:
    #    - Pop (d, u). If d > dist[u], skip (stale).
    #    - For each (v, w) in graph[u]:
    #        if dist[u] + w < dist[v]: update dist[v], parent[v], push
    pass
```

```python
# Test on Dijkstra graph from warm-up
# s=0, a=1, b=2, c=3, d=4
graph_dijk = [[] for _ in range(5)]
dijk_edges = [(0,1,10), (0,2,5), (1,3,1), (1,2,2), (2,1,3),
              (2,4,9), (2,3,2), (3,4,4), (4,0,6)]
for u, v, w in dijk_edges:
    graph_dijk[u].append((v, w))

dist, parent = dijkstra(graph_dijk, 0)
print(f"Distances: {dist}")
print(f"Parents:   {parent}")

# Expected: s=0, a=8, b=5, c=7, d=11
assert dist == [0, 8, 5, 7, 11]

path_to_d = reconstruct_path(parent, 0, 4)
print(f"Path s->d: {path_to_d}")
# Expected path: s -> b -> c -> d (0 -> 2 -> 3 -> 4)
print("Dijkstra test passed!")
```

8 Negative Cycles & Algorithm Limits (10 min)

8.1 Detecting negative cycles with Bellman-Ford

```python
# Graph with a negative cycle: 0 -> 1 -> 2 -> 0
#   0 -> 1 (1), 1 -> 2 (2), 2 -> 0 (-5)
#   0 -> 3 (4)
edges_neg = [
    (0, 1, 1),
    (1, 2, 2),
    (2, 0, -5),  # negative cycle: 1 + 2 + (-5) = -2
    (0, 3, 4),
]

dist, parent, neg_cycle = bellman_ford(4, edges_neg, 0)
print(f"Negative cycle detected: {neg_cycle}")
assert neg_cycle == True
print("Negative cycle detection test passed!")
```

8.2 Why Dijkstra fails with negative weights

```python
# Counter-example: s -> a (1), s -> b (3), b -> a (-4)
# True shortest path to a: s -> b -> a = 3 + (-4) = -1
# Dijkstra will finalize a with d=1 (via direct edge s->a)
graph_neg = [[], [], []]
graph_neg[0] = [(1, 1), (2, 3)]  # s -> a(1), s -> b(3)
graph_neg[2] = [(1, -4)]         # b -> a(-4)

dist_dijk, _ = dijkstra(graph_neg, 0)
dist_bf, _, _ = bellman_ford(3, [(0,1,1), (0,2,3), (2,1,-4)], 0)

print(f"Dijkstra distances:    {dist_dijk}")
print(f"Bellman-Ford distances: {dist_bf}")
print(f"Dijkstra gives a.d = {dist_dijk[1]}, "
      f"but true shortest is {dist_bf[1]}")
```

Question: Why does Dijkstra give the wrong answer here? Which step in the correctness proof fails?

9 Performance Comparison 💻 (10 min)

9.1 Kruskal vs. Prim on random graphs

```python
import time
import random

def random_connected_graph(n, extra_edges):
    """Generate a random connected weighted graph."""
    edges = []
    # Spanning tree to ensure connectivity
    for i in range(1, n):
        j = random.randint(0, i - 1)
        w = random.randint(1, 100)
        edges.append((w, i, j))
    # Extra edges
    for _ in range(extra_edges):
        u, v = random.sample(range(n), 2)
        w = random.randint(1, 100)
        edges.append((w, u, v))
    return edges

def edges_to_adj(n, edges):
    """Convert edge list to adjacency list."""
    graph = [[] for _ in range(n)]
    for w, u, v in edges:
        graph[u].append((v, w))
        graph[v].append((u, w))
    return graph

sizes = [100, 500, 1000, 2000, 5000]
print(f"{'n':>6} | {'Kruskal':>12} | {'Prim':>12}")
print("-" * 36)

for n in sizes:
    edges = random_connected_graph(n, 3 * n)
    graph = edges_to_adj(n, edges)

    start = time.perf_counter()
    mst_k = kruskal(n, edges[:])
    t_kruskal = time.perf_counter() - start

    start = time.perf_counter()
    mst_p = prim(graph, 0)
    t_prim = time.perf_counter() - start

    w_k = sum(w for _, _, w in mst_k)
    w_p = sum(w for _, _, w in mst_p)
    assert w_k == w_p, f"MST weights differ: {w_k} vs {w_p}"

    print(f"{n:>6} | {t_kruskal:>12.6f} | {t_prim:>12.6f}")

print("Performance comparison passed!")
```

Questions:

(a) Which algorithm is faster on sparse graphs ($E = \Theta(V)$)?

(b) What are the asymptotic complexities of each?

9.2 Dijkstra vs. Bellman-Ford

```python
def random_directed_graph(n, num_edges):
    """Generate a random directed graph with nonneg. weights."""
    edges = []
    # Ensure connectivity: chain 0 -> 1 -> ... -> n-1
    for i in range(n - 1):
        w = random.randint(1, 50)
        edges.append((i, i + 1, w))
    for _ in range(num_edges - (n - 1)):
        u, v = random.sample(range(n), 2)
        w = random.randint(1, 50)
        edges.append((u, v, w))
    return edges

def edges_to_directed_adj(n, edges):
    graph = [[] for _ in range(n)]
    for u, v, w in edges:
        graph[u].append((v, w))
    return graph

sizes = [100, 500, 1000, 2000]
print(f"\n{'n':>6} | {'Dijkstra':>12} | {'Bellman-Ford':>12}")
print("-" * 36)

for n in sizes:
    edges = random_directed_graph(n, 4 * n)
    graph = edges_to_directed_adj(n, edges)

    start = time.perf_counter()
    dist_d, _ = dijkstra(graph, 0)
    t_dijk = time.perf_counter() - start

    start = time.perf_counter()
    dist_bf, _, _ = bellman_ford(n, edges, 0)
    t_bf = time.perf_counter() - start

    # Verify both give same results
    assert dist_d == dist_bf, "Dijkstra and Bellman-Ford disagree!"

    print(f"{n:>6} | {t_dijk:>12.6f} | {t_bf:>12.6f}")

print("Dijkstra vs Bellman-Ford comparison passed!")
```

Question: What speedup does Dijkstra achieve over Bellman-Ford? Is this consistent with the theoretical complexities $\mathcal{O}(E \log V)$ vs. $\mathcal{O}(VE)$?

Bonus: DAG Shortest Paths

Bonus

Implement shortest paths in a DAG using topological sort. This runs in $\Theta(V+E)$ — faster than both Bellman-Ford and Dijkstra.

(a) Implement `dag_shortest_paths(graph, s)` that:

1. Topologically sorts the graph
2. Initializes distances

3. Processes vertices in topological order, relaxing outgoing edges

(b) Test it on a DAG of your choice. Verify against Bellman-Ford.

(c) **Application:** given a DAG with edge weights representing task durations, find the **critical path** (longest path). *Hint:* negate all weights and run DAG shortest paths.