1 Warm-up: MST by Hand

1.1 Kruskal's algorithm

Solution

(a) Sorted edges: $(2,3):1$, $(4,5):2$, $(1,2):3$, $(5,6):4$, $(1,6):5$, $(3,4):6$, $(2,6):7$, $(3,5):8$.
(b) Step-by-step execution:

Step	Edge	Action	Components
–	–	init	$\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}$
1	$(2,3):1$	add	$\{1\},\{2,3\},\{4\},\{5\},\{6\}$
2	$(4,5):2$	add	$\{1\},\{2,3\},\{4,5\},\{6\}$
3	$(1,2):3$	add	$\{1,2,3\},\{4,5\},\{6\}$
4	$(5,6):4$	add	$\{1,2,3\},\{4,5,6\}$
5	$(1,6):5$	add	$\{1,2,3,4,5,6\}$

Edges $(3,4):6$, $(2,6):7$, $(3,5):8$ would form cycles and are not considered (we already have $n-1=5$ edges).
(c) MST $=\{(2,3),(4,5),(1,2),(5,6),(1,6)\}$ with total weight $1+2+3+4+5=$ **15**.

Common Mistakes

- Students confuse "skip" (both endpoints same component) and "add".
- Forgetting to stop after $n-1$ edges.
- Arithmetic errors in total weight.

1.2 Prim's algorithm

Solution

Starting from vertex 1. V_T is the set of vertices in the tree.

Step	Extract	Edge added	PQ updates
0	1 (key=0)	–	2:3, 6:5
1	2 (key=3)	$(1,2):3$	3:1, 6: $\min(5,7)$=5
2	3 (key=1)	$(2,3):1$	4:6, 5:8
3	6 (key=5)	$(1,6):5$	5: $\min(8,4)$=4
4	5 (key=4)	$(6,5):4$	4: $\min(6,2)$=2
5	4 (key=2)	$(5,4):2$	–

Detail: After extracting vertex 3, the PQ contains 6:5, 4:6, 5:8. The minimum key is 6:5, so vertex 6 is extracted next (not vertex 4). Vertex 6's neighbor 5 (not yet in tree) gets key $\min(8,4)=4$.

Then vertex 5 is extracted (key=4), and its neighbor 4 gets key $\min(6, 2) = 2$. Finally vertex 4 is extracted (key=2).
MST edges: $(1, 2) : 3$, $(2, 3) : 1$, $(1, 6) : 5$, $(6, 5) : 4$, $(5, 4) : 2$. Total: $3 + 1 + 5 + 4 + 2 = \mathbf{15}$.
(b) Both Kruskal and Prim produce MSTs with the same total weight **15**. In fact the edge sets are identical here: $\{(2, 3), (4, 5), (1, 2), (5, 6), (1, 6)\}$, discovered in different order. In general, when edge weights are distinct, the MST is unique; when there are ties, different algorithms may select different edges but the total weight is always the same.

Grading Notes

Full marks for correct step-by-step trace with components/PQ shown. Accept either order for tied edges. Both must give weight 15.

2 Implement Union-Find

Solution

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        rx, ry = self.find(x), self.find(y)
        if rx == ry:
            return False
        if self.rank[rx] < self.rank[ry]:   # union by rank
            rx, ry = ry, rx
        self.parent[ry] = rx
        if self.rank[rx] == self.rank[ry]:
            self.rank[rx] += 1
        return True
```

Key points:

- **Path compression** (`find`): makes every node on the find path point directly to the root.
- **Union by rank**: attach the shorter tree under the root of the taller tree.
- Combined, m operations on n elements take $\mathcal{O}(m \cdot \alpha(n))$ — nearly $\mathcal{O}(m)$.

Common Mistakes

- **Forgetting path compression:** correct but degrades to $\mathcal{O}(n)$ per find.
- **Comparing `x` and `y` instead of `rx` and `ry`**: must compare roots.
- **Incrementing rank on every union:** rank should only increase when both trees have equal rank.

3 Implement Kruskal's Algorithm

Solution

```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
    """
    edges.sort()
    uf = UnionFind(n)
    mst = []
    for w, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, w))
            if len(mst) == n - 1:
                break
    return mst
```

Output:

```
Kruskal MST: [(1, 2, 1), (3, 4, 2), (0, 1, 3), (4, 5, 4), (5, 0, 5)]
Total weight: 15
Kruskal test passed!
```

Note: vertices are 0-indexed (0=1, 1=2, ..., 5=6 from the graph).

Explanation

Complexity: $\mathcal{O}(E \log E)$ for sorting $+\mathcal{O}(E \cdot \alpha(V))$ for Union-Find operations $= \mathcal{O}(E \log V)$ total (since $\log E = \Theta(\log V)$ when $E \leqslant V^2$).
The early-exit optimization (`if len(mst) == n-1: break`) saves time in practice but doesn't change the worst-case complexity (sorting dominates).

4 Implement Prim's Algorithm

Solution

```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
    """
    n = len(graph)
    key = [float('inf')] * n
    parent = [None] * n
    in_tree = [False] * n
    key[r] = 0
    pq = [(0, r)]  # (key, vertex)
    mst = []

    while pq:
        k, u = heapq.heappop(pq)
        if in_tree[u]:
            continue
        in_tree[u] = True
        if parent[u] is not None:
            mst.append((parent[u], u, k))
        for v, w in graph[u]:
            if not in_tree[v] and w < key[v]:
                key[v] = w
                parent[v] = u
                heapq.heappush(pq, (w, v))
    return mst
```

Output:

```
Prim MST: [(0, 1, 3), (1, 2, 1), (0, 5, 5), (5, 4, 4), (4, 3, 2)]
Total weight: 15
Prim test passed!
```

Common Mistakes

- **Not skipping stale PQ entries:** the `if in_tree[u]: continue` check is essential with lazy deletion.
- **Using dist instead of key:** Prim's key is the edge weight to the tree, *not* the total distance from root.
- **Confusing Prim with Dijkstra:** the key update is `w < key[v]` (Prim) vs. `dist[u] + w < dist[v]` (Dijkstra).

Grading Notes

Both Kruskal and Prim must produce MST weight = 15. Edge order within the MST may differ. Accept any valid MST.

5 Warm-up: Shortest Paths by Hand

5.1 Bellman-Ford

Solution

Graph: $s \to a$ (6), $s \to c$ (4), $a \to b$ (-2), $a \to c$ (3), $c \to d$ (5), $b \to d$ (1), $c \to a$ (-3).
Using edge order: (s,a), (s,c), (a,b), (a,c), (c,d), (b,d), (c,a).
(a) Distance estimates after each pass:

	s	a	b	c	d
Init	0	∞	∞	∞	∞
Pass 1	0	1	4	4	5
Pass 2	0	1	-1	4	0
Pass 3	0	1	-1	4	0
Pass 4	0	1	-1	4	0

Detailed pass 1: (edge order: (s,a), (s,c), (a,b), (a,c), (c,d), (b,d), (c,a); updates are *immediate*)

- Relax (s,a): $a.d = \min(\infty, 0+6) = 6$.
- Relax (s,c): $c.d = \min(\infty, 0+4) = 4$.
- Relax (a,b): $b.d = \min(\infty, 6+(-2)) = 4$.
- Relax (a,c): $c.d = \min(4, 6+3) = 4$ (no change).
- Relax (c,d): $d.d = \min(\infty, 4+5) = 9$.
- Relax (b,d): $d.d = \min(9, 4+1) = 5$.
- Relax (c,a): $a.d = \min(6, 4+(-3)) = 1$.

After pass 1: $d = [0, 1, 4, 4, 5]$.
Pass 2:

- (s,a), (s,c): no change.
- (a,b): $b.d = \min(4, 1+(-2)) = -1$.
- (a,c), (c,d): no change.
- (b,d): $d.d = \min(5, (-1)+1) = 0$.
- (c,a): no change.

After pass 2: $d = [0, 1, -1, 4, 0]$.
Pass 3–4: No changes (converged).
(b) No negative-weight cycle: pass 4 produces no changes, and the check pass finds no edge that can be further relaxed.
(c) Shortest-path tree (π values): $\pi[a] = c$, $\pi[b] = a$, $\pi[c] = s$, $\pi[d] = b$.
Shortest paths:

- $s \to a$: $s \to c \to a$ (weight $4+(-3) = 1$).
- $s \to b$: $s \to c \to a \to b$ (weight $4+(-3)+(-2) = -1$).
- $s \to c$: $s \to c$ (weight 4).
- $s \to d$: $s \to c \to a \to b \to d$ (weight $4+(-3)+(-2)+1 = 0$).

Common Mistakes

- **Wrong edge relaxation order:** the results after each pass depend on the order edges are processed. Any valid order is acceptable, but consistency is required.
- **Confusing "converged" with "negative cycle":** convergence in fewer than $|V|-1$ passes just means shortest paths were found early. A negative cycle is only detected if the $(|V|)$-th pass can still relax.
- **Forgetting about edge $c \to a$ with weight -3:** this edge is crucial and creates the path $s \to c \to a$ with weight 1 (shorter than the direct $s \to a$ with weight 6).

5.2 Dijkstra

Solution

Graph: $s \to a$ (10), $s \to b$ (5), $a \to c$ (1), $a \to b$ (2), $b \to a$ (3), $b \to d$ (9), $b \to c$ (2), $c \to d$ (4), $d \to s$ (6).

(a) Step-by-step:

Step	Extract	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$
Init	–	0	∞	∞	∞	∞
1	s (d=0)	0	10	5	∞	∞
2	b (d=5)	0	8	5	7	14
3	c (d=7)	0	8	5	7	11
4	a (d=8)	0	8	5	7	11
5	d (d=11)	0	8	5	7	11

Detailed:

- Extract s (d=0): relax $s \to a$ ($d[a] = 10$), $s \to b$ ($d[b] = 5$).
- Extract b (d=5): relax $b \to a$ ($d[a] = \min(10, 5+3) = 8$), $b \to c$ ($d[c] = 5+2 = 7$), $b \to d$ ($d[d] = 5+9 = 14$).
- Extract c (d=7): relax $c \to d$ ($d[d] = \min(14, 7+4) = 11$).
- Extract a (d=8): relax $a \to c$ ($d[c] = \min(7, 8+1) = 7$, no change), $a \to b$ ($d[b] = \min(5, 8+2) = 5$, no change).
- Extract d (d=11): relax $d \to s$ ($d[s] = \min(0, 11+6) = 0$, no change).

(b) Shortest-path tree: $\pi[a] = b$, $\pi[b] = s$, $\pi[c] = b$, $\pi[d] = c$.

(c) Shortest path $s \to d$: $s \to b \to c \to d$, weight $= 5+2+4 = 11$.

Grading Notes

Full marks require: (1) correct extraction order, (2) correct d-values after each extraction, (3) valid shortest-path tree.

6 Implement Bellman-Ford

Solution

```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)
    """
    dist = [float('inf')] * n
    parent = [None] * n
    dist[s] = 0

    for _ in range(n - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u

    # Check for negative-weight cycles
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            return dist, parent, True  # negative cycle

    return dist, parent, False
```

Output:

```
Distances: [0, 1, -1, 4, 0]
Parents:   [None, 3, 1, 0, 2]
Negative cycle: False
Path s->d: [0, 3, 1, 2, 4]
Bellman-Ford test passed!
```

Path $s \to d$: $0 \to 3 \to 1 \to 2 \to 4$, i.e., $s \to c \to a \to b \to d$, weight $4 + (-3) + (-2) + 1 = 0$.

Explanation

Complexity: $\mathcal{O}(VE)$ — the outer loop runs $|V| - 1$ times, each iterating over all $|E|$ edges. **Early termination:** if no edge is relaxed during an entire pass, the algorithm can stop early (all distances are final). This doesn't change the worst case but helps in practice.

Common Mistakes

- **Using `dist[u] != inf and dist[u] + w < dist[v]`:** the `!= inf` guard is unnecessary in Python (since `inf + w = inf`), but it's a common pattern from C/C++ implementations.
- **Off-by-one in loop count:** the main loop must run exactly $n - 1$ times, not n.
- **Forgetting the negative cycle check:** the final pass is essential — without it, the algorithm silently returns wrong results when a negative cycle exists.

7 Implement Dijkstra's Algorithm

```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)
    """
    n = len(graph)
    dist = [float('inf')] * n
    parent = [None] * n
    dist[s] = 0
    pq = [(0, s)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue  # stale entry
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u
                heapq.heappush(pq, (dist[v], v))

    return dist, parent
```

Output:

```
Distances: [0, 8, 5, 7, 11]
Parents:   [None, 2, 0, 2, 3]
Path s->d: [0, 2, 3, 4]
Dijkstra test passed!
```

Path $s \to d$: $0 \to 2 \to 3 \to 4$, i.e., $s \to b \to c \to d$, weight $5 + 2 + 4 = 11$.

Explanation

Lazy deletion: instead of implementing `decrease_key` (which `heapq` doesn't support natively), we push a new entry for every relaxation and skip stale ones. This means the heap can grow to $\mathcal{O}(E)$ entries, but the total work is still $\mathcal{O}(E \log V)$ since each edge generates at most one push.
Alternative: use an indexed priority queue or the `heapdict` library for a true $\mathcal{O}(V \log V + E)$ Fibonacci-heap-like behavior.

Common Mistakes

- **Missing the stale entry check:** without `if d > dist[u]: continue`, the algorithm processes vertices multiple times and becomes slower (though still correct for nonneg. weights).
- **Using Dijkstra with negative weights:** produces incorrect results. Always check the problem constraints.
- **Confusing Prim and Dijkstra:** Prim uses edge weight as key; Dijkstra uses total distance.

8 Negative Cycles & Algorithm Limits

8.1 Detecting negative cycles with Bellman-Ford

Solution

Output:

```
Negative cycle detected: True
Negative cycle detection test passed!
```

The cycle $0 \to 1 \to 2 \to 0$ has total weight $1 + 2 + (-5) = -2 < 0$. After 3 passes ($|V| - 1 = 3$), the check pass can still relax at least one edge, indicating a negative cycle.

8.2 Why Dijkstra fails with negative weights

Solution

Output:

```
Dijkstra distances:    [0, 1, 3]
Bellman-Ford distances: [0, -1, 3]
Dijkstra gives a.d = 1, but true shortest is -1
```

Explanation: Dijkstra extracts s (d=0), updates $a.d = 1$ and $b.d = 3$. Then it extracts a (d=1) and *finalizes* it. Later, when b is extracted (d=3), relaxing $b \to a$ would give $a.d = 3 + (-4) = -1 < 1$, but a is already finalized.
Which proof step fails? In the correctness proof, step 4 requires $\delta(s, y) \leqslant \delta(s, u)$ for any vertex y on the shortest path between s and the extracted vertex u. This relies on *all weights being nonnegative*. With the negative edge $b \to a$ ($w = -4$), we have $\delta(s, a) = -1 < \delta(s, b) = 3$, but a is extracted before b.

9 Performance Comparison

9.1 Kruskal vs. Prim on random graphs

Solution

Typical output:

```
     n |      Kruskal |         Prim
-------------------------------------
   100 |     0.000134 |     0.000089
   500 |     0.001023 |     0.000612
  1000 |     0.002341 |     0.001478
  2000 |     0.005602 |     0.003215
  5000 |     0.016234 |     0.009876
```

(a) For sparse graphs ($E = \Theta(V)$), Prim with a binary heap is typically faster due to lower constant factors (heapq is C-implemented in Python, whereas Kruskal's sorting has Python-level overhead).
(b) Asymptotic complexities:

- Kruskal: $\mathcal{O}(E \log V)$ (dominated by sorting).
- Prim (binary heap): $\mathcal{O}(E \log V)$.

Both are $\mathcal{O}(E \log V)$ for sparse graphs. The differences are due to constant factors.

9.2 Dijkstra vs. Bellman-Ford

Solution

Typical output:

```
     n |     Dijkstra | Bellman-Ford
-----------------------------------
   100 |     0.000051 |     0.000423
   500 |     0.000312 |     0.011234
  1000 |     0.000798 |     0.048234
  2000 |     0.001856 |     0.198234
```

Observations:

- Dijkstra is **10–100x faster** than Bellman-Ford on these graphs.
- Dijkstra scales as $\mathcal{O}(E \log V)$; Bellman-Ford as $\mathcal{O}(VE)$.
- For $E = 4V$: Dijkstra is $\mathcal{O}(V \log V)$, Bellman-Ford is $\mathcal{O}(V^2)$.
- The speedup ratio grows with n, consistent with a factor of $\mathcal{O}(V/\log V)$.

Conclusion: use Dijkstra whenever possible (nonneg. weights). Use Bellman-Ford only when negative edges are present.

Grading Notes

Students should observe that: (1) Kruskal and Prim have similar performance, (2) Dijkstra is significantly faster than Bellman-Ford, and (3) the speedup grows with graph size. Exact numbers will vary.

Bonus: DAG Shortest Paths

Solution

```python
from collections import deque

def topological_sort(graph):
    """Kahn's algorithm for topological sort."""
    n = len(graph)
    in_degree = [0] * n
    for u in range(n):
        for v, _ in graph[u]:
            in_degree[v] += 1
    queue = deque(v for v in range(n) if in_degree[v] == 0)
    order = []
    while queue:
        u = queue.popleft()
        order.append(u)
        for v, _ in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)
    return order


def dag_shortest_paths(graph, s):
    """Shortest paths in a DAG in O(V + E)."""
    n = len(graph)
    dist = [float('inf')] * n
    parent = [None] * n
    dist[s] = 0

    order = topological_sort(graph)
    for u in order:
        if dist[u] == float('inf'):
            continue
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                parent[v] = u

    return dist, parent
```

Test:

```python
# DAG: 0 -> 1(5), 0 -> 2(3), 1 -> 3(6), 1 -> 2(2), 2 -> 3(7),
#       2 -> 4(4), 3 -> 4(-1), 3 -> 5(1), 4 -> 5(-2)
dag = [[(1,5),(2,3)], [(3,6),(2,2)], [(3,7),(4,4)],
       [(4,-1),(5,1)], [(5,-2)], []]

dist_dag, _ = dag_shortest_paths(dag, 0)
dist_bf, _, _ = bellman_ford(6,
    [(0,1,5),(0,2,3),(1,3,6),(1,2,2),(2,3,7),(2,4,4),(3,4,-1),(3,5,1),(4,5,-2)], 0)

assert dist_dag == dist_bf
print(f"DAG distances: {dist_dag}")
# Expected: [0, 5, 3, 10, 7, 5]
```

Critical path application: negate all edge weights and run `dag_shortest_paths`. The negation of the resulting distance gives the longest path (critical path).

```python
def critical_path(graph, s):
    """Find longest path in DAG (critical path)."""
    # Negate weights
    neg_graph = [[(v, -w) for v, w in adj] for adj in graph]
    dist, parent = dag_shortest_paths(neg_graph, s)
    return [-d for d in dist], parent
```

Explanation

Complexity: $\Theta(V + E)$ — topological sort is $\Theta(V + E)$, and the relaxation pass visits each vertex and edge exactly once. This is optimal and faster than both Dijkstra ($\mathcal{O}(E \log V)$) and Bellman-Ford ($\mathcal{O}(VE)$), but only works for DAGs.
Negative weights: unlike Dijkstra, DAG shortest paths correctly handles negative weights (no cycles to worry about in a DAG).