This assignment is about implementing Kruskal's and Prim-Dijkstra's MST finding algorithms and comparing their execution times on random graphs of varying sizes (number of nodes) and density (number of edges). The report is organized as follows: source code, brief description about the experimental setup, comparison plot, graph generation, and data structures.

Source code

The source code for the implementation of Kruskal's and Prim-Dijkstra's MST finding algorithms with varying inputs to generate the comparison plot along with the required data structures is given below:

```
# --------------------------
# Imports
# --------------------------

import random
import time
import heapq
import matplotlib.pyplot as plt
from collections import defaultdict

# --------------------------
# Graph Generator
# --------------------------

def generate_connected_random_graph(n, m):
    """
    Generates a connected undirected weighted graph with n nodes and m
edges.
    Ensures initial connectivity by forming a linear chain, then adds
extra edges randomly.
    """
    assert m >= n - 1
    W = [[0 for _ in range(n)] for _ in range(n)]  # Adjacency matrix
    edges = []  # Edge list: (u, v, weight)

    # Step 1: Linear chain to guarantee connectivity
    for i in range(n - 1):
        w = random.randint(1, 1000)
        W[i][i + 1] = W[i + 1][i] = w
        edges.append((i, i + 1, w))


    # Step 2: Add remaining random edges
    count = n - 1
    while count < m:
        i, j = random.randint(0, n - 1), random.randint(0, n - 1)
        if i != j and W[i][j] == 0:
            w = random.randint(1, 1000)
            W[i][j] = W[j][i] = w
            edges.append((i, j, w))
            count += 1

    return W, edges


# ---------------------------
# Kruskal's Algorithm
# ---------------------------

def find(parent, i):
    # Path compression for Union-Find
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]

def union(parent, rank, x, y):
    # Union by rank to optimize merging
    xroot = find(parent, x)
    yroot = find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1


def kruskal(n, edges):
    """
    Kruskal's MST algorithm using Union-Find and sorted edge list.
    Returns total weight of the MST.
    """
```

```python
    parent = list(range(n))
    rank = [0] * n
    edges.sort(key=lambda x: x[2])  # Sort edges by weight

    mst_weight = 0
    edge_count = 0

    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            union(parent, rank, u, v)
            mst_weight += w
            edge_count += 1
            if edge_count == n - 1:
                break

    return mst_weight


# --------------------------
# Prim's Algorithm
# --------------------------

def prim(n, W):
    """
    Prim's MST algorithm using binary min-heap (heapq).
    Operates on adjacency matrix W.
    Returns total weight of the MST.
    """
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, node)
    mst_weight = 0

    while min_heap:
        w, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        mst_weight += w
        for v in range(n):
            if W[u][v] != 0 and not visited[v]:
                heapq.heappush(min_heap, (W[u][v], v))

    return mst_weight
```

```python
# ---------------------------
# Runtime Measurement
# ---------------------------

def measure_time(func, *args):
    """
    Measures the runtime of a given function.
    Returns: (result of function, execution time in seconds)
    """
    start = time.time()
    result = func(*args)
    end = time.time()
    return result, (end - start)


# ---------------------------
# Running the Experiment
# ---------------------------

def run_experiment():
    """
    Runs MST algorithm experiments across different graph sizes and densities.
    Repeats 5 times for each (n, m/n) to compute average execution times.
    """
    sizes = [16, 32, 64, 128, 256]
    densities = [1, 2, 4, 8, 16, 32]
    results = {"kruskal": {}, "prim": {}}

    for n in sizes:
        for d in densities:
            m = min(d * n, n * (n - 1) // 2)  # Upper bound: complete graph

            kruskal_times = []
            prim_times = []

            for _ in range(5):  # Repeat 5 times per configuration
                W, edges = generate_connected_random_graph(n, m)
                _, k_time = measure_time(kruskal, n, edges)
                _, p_time = measure_time(prim, n, W)
                kruskal_times.append(k_time)
                prim_times.append(p_time)

            # Compute average runtime
            avg_k = sum(kruskal_times) / 5
            avg_p = sum(prim_times) / 5
            results["kruskal"].setdefault(n, []).append((d, avg_k))
            results["prim"].setdefault(n, []).append((d, avg_p))

    return results


# --------------------------
# Plotting Results
# --------------------------

def plot_results(results):
    """
    Plots average execution time vs. density (m/n) for each algorithm and graph size.
    """
    plt.figure(figsize=(12, 7))
    for algo in results:
        for n in results[algo]:
            xs = [d for d, _ in results[algo][n]]
            ys = [t for _, t in results[algo][n]]
            plt.plot(xs, ys, marker='o', label=f"{algo.capitalize()} ({n})")

    plt.xscale("log", base=2)
    plt.xlabel("Density (m/n)")
    plt.ylabel("Average Execution Time (s)")
    plt.title("Kruskal vs Prim Execution Time on Random Graphs")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()


# --------------------------
# Main Execution
# --------------------------

results = run_experiment()
plot_results(results)
```

Experimental Setup

Processor: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s)

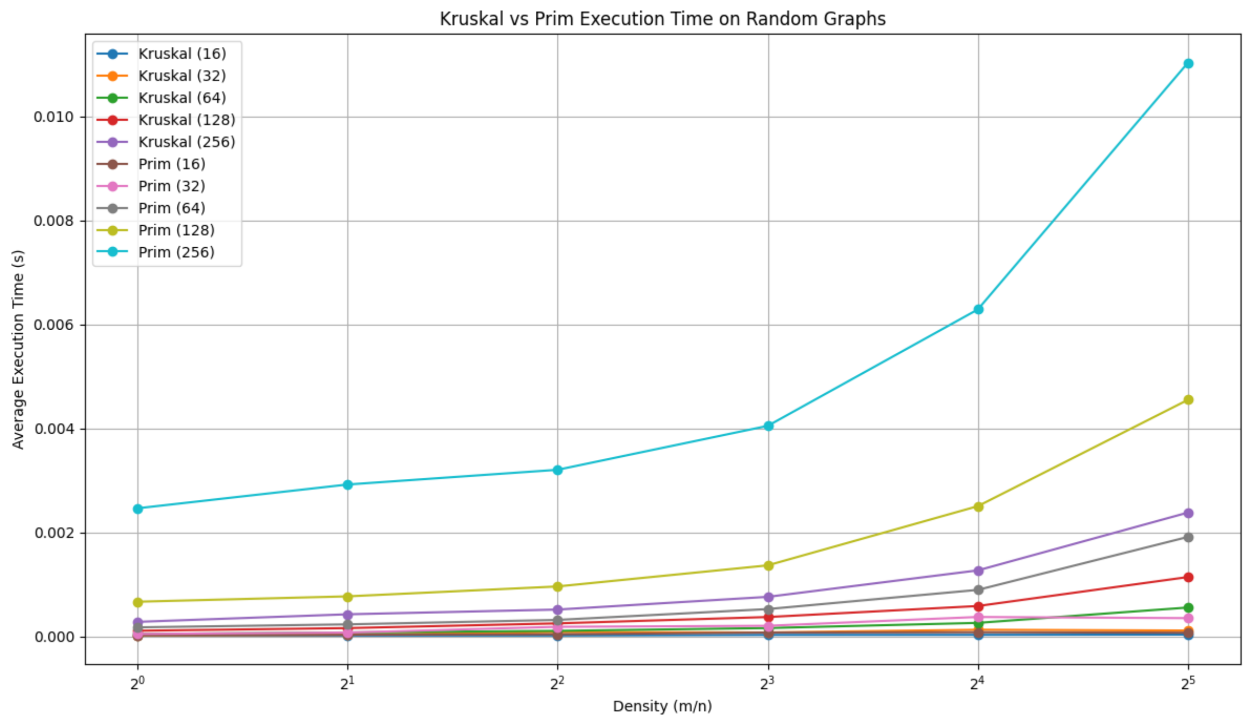
Language: Python

Random Number Generator: random.randint() function using random library in Python.

Comparison Plot

The objective of the plot was to:

- Compare Kruskal's and Prim's MST algorithms.
- Measure their execution times across different graph sizes (n) and densities (m/n).
- Plot average execution time vs. density, with:
    - X-axis = m/n (log scale)
    - Y-axis = execution time in seconds
    - One curve per (n, algorithm) pair



The selected values of n are 16, 32, 64, 128, 256. For each n, the varied m (number of edges) as m = n, 2n, 4n, 8n, 16n, 32n up to complete graph: n(n-1)/2. For each (n, m) pair, 5 random connected graphs were generated. The execution time of both Kruskal's algorithm (using edge list + union-find) and Prim's algorithm (using adjacency matrix + binary heap) were measured and averaged. The plot was done by keeping m/n (density) on the X-axis and execution time on the Y-axis. Each curve is labeled by algorithm and size, e.g., (128, Kruskal).

X-axis (Density m/n):

- Scaled logarithmically with base 2.
- Shows how graph density increases exponentially (1, 2, 4, ..., 32).
- Allows fair visual spacing for rapid growth in edges.

Y-axis (Average Execution Time in seconds):

- Linear scale.
- Plots the actual measured execution time.

The provided plot illustrates the comparative performance of Kruskal's and Prim's minimum spanning tree (MST) algorithms across varying graph sizes and densities. Graph sizes (n) range from 16 to 256, and for each size, edge counts (m) are scaled to reflect increasing density, with the x-axis representing m/n on a logarithmic scale. The y-axis shows the average execution time in seconds, computed from five random connected graphs for each (n, m) pair. Graph generation time was excluded to ensure accurate measurement of algorithm performance.

Each curve corresponds to a fixed n and algorithm, allowing direct visual comparison. The results show that Kruskal's algorithm scales more efficiently with increasing density, particularly in larger graphs, due to its O(E log E) complexity and edge-focused approach. In contrast, Prim's algorithm, implemented with an adjacency matrix and binary heap, demonstrates steeper growth in execution time as both n and m increase which is especially evident for n = 256. It highlights its sensitivity to dense connectivity. Overall, Kruskal outperforms Prim in dense and large graphs, while both perform comparably in smaller or sparser scenarios.

Graph Generation

To evaluate the performance of Minimum Spanning Tree (MST) algorithms such as Kruskal's and Prim's, a consistent and controlled method of graph generation is necessary. The process ensures that the graph is always connected while allowing for randomization in edge weights. The task specifies different strategies depending on the density of the graph, governed by the number of vertices n and edges m.

```python
def graph_generation():
    sizes = [16, 32, 64, 128, 256]
    densities = [1, 2, 4, 8, 16, 32]
    results = {"kruskal": {}, "prim": {}}

    for n in sizes:
        for d in densities:
            m = min(d * n, n * (n - 1) // 2)
            kruskal_times = []
            prim_times = []

            for _ in range(5):  # Repeat 5 times per configuration
                W, edges = generate_connected_random_graph(n, m)
                _, k_time = measure_time(kruskal, n, edges)
                _, p_time = measure_time(prim, n, W)
                kruskal_times.append(k_time)
                prim_times.append(p_time)

            # Compute average runtime
            avg_k = sum(kruskal_times) / 5
            avg_p = sum(prim_times) / 5
            results["kruskal"].setdefault(n, []).append((d, avg_k))
            results["prim"].setdefault(n, []).append((d, avg_p))

    return results
```

General Graph Generation Process (For any n and m):

Initialization of the Weight Matrix

The generation begins by initializing an n × n weight matrix W filled with zeros. Each cell W[i][j] represents the weight of an edge from vertex i to vertex j. A value of 0 means that no edge currently exists between those two nodes.

Example: When n = 16, the matrix W is a 16×16 matrix initialized as:

```python
W = [[0 for _ in range(n)] for _ in range(n)]
```

This structure allows for efficient lookup and update of weights and will be used to construct both sparse and dense graphs.

Ensuring Connectivity with a Spanning Path

To guarantee that the graph is connected, a deterministic set of n - 1 edges is inserted. These edges form a linear path from node 1 to node n.

Example: For n = 16, the following 15 edges are added:

$$\{1, 2\}, \{2, 3\}, \{3, 4\}, ..., \{15, 16\}$$

In zero-indexed code, this translates to:

```
for i in range(n - 1):
    w = random.randint(1, 1000)
    W[i][i + 1] = w
    W[i + 1][i] = w
```

This step ensures that the graph is fully connected from the start, and no part of the graph is isolated.

Adding Random Edges (When m > n - 1)

Once the base connectivity is established, the remaining m - (n - 1) edges must be added randomly. For each new edge:

- Two random integers i and j are chosen such that $0 \leq i < j < n$.
- A random weight w is generated within the range 1 to 1000.
- The edge {i, j} is only added if it does not already exist in the matrix, i.e., W[i][j] == 0.

This process repeats until exactly m unique edges are present in the graph.

Example: If n = 64 and m = 512 (i.e., density = 8, as m/n = 8), the base path contributes 63 edges. Therefore, 512 - 63 = 449 additional random edges must be generated.

The corresponding code snippet might look like:

```
added = set()
while len(added) < (m - (n - 1)):
    i = random.randint(0, n - 1)
    j = random.randint(0, n - 1)
    if i != j and W[i][j] == 0:
        w = random.randint(1, 1000)
        W[i][j] = w
        W[j][i] = w
        added.add((min(i, j), max(i, j)))
```

This ensures no duplication and maintains symmetry for the undirected graph.

Generating a Complete Graph Efficiently (m = n(n - 1)/2)

For this special case, the graph must contain every possible pair of nodes, which equates to n(n - 1)/2 edges. Attempting to add edges one by one while checking for duplicates would be highly inefficient.

Instead, the method directly populates the upper triangle of the matrix with unique random weights.

Example: For n = 16, the total number of edges is:

$$\frac{16 \times 15}{2} = 120$$

A total of 120 unique random weights are generated and assigned as follows:

```
weights = random.sample(range(1, 1001), k=(n * (n - 1)) // 2)
idx = 0
for i in range(n):
    for j in range(i + 1, n):
        W[i][j] = weights[idx]
        W[j][i] = weights[idx]
        idx += 1
```

This guarantees full connectivity, symmetric weights, and avoids repeated checking or retries.

Optimizing Generation for Dense Graphs

For dense graphs, where m is large (e.g., m = 992 for n = 64), generating and validating random edges becomes inefficient due to high collision probability in the weight matrix. Instead, it is more practical to begin with a complete graph and delete random edges until the count reaches m.

Procedure:

- Generate a complete graph as above.
- Create a list of all edge pairs excluding the deterministic path {i, i+1} for all i.
- Randomly delete edges from the list while ensuring that none of the critical path edges are removed.

Example: If n = 64, the complete graph contains 2016 edges. If the target m = 1024, then 2016 - 1024 = 992 edges must be deleted.

```
critical_edges = {(i, i + 1) for i in range(n - 1)}
all_edges = {(i, j) for i in range(n) for j in range(i + 1, n)}
deletable = list(all_edges - critical_edges)
to_delete = random.sample(deletable, len(all_edges) - m)

for (i, j) in to_delete:
    W[i][j] = 0
    W[j][i] = 0
```

This ensures that:

- The graph remains connected via the preserved path.
- Edge count matches the desired m.
- Randomness is preserved for the remaining structure.

Data Structures

Kruskal’s Algorithm

Kruskal’s algorithm builds a minimum spanning tree (MST) by greedily choosing the smallest weight edge at each step, ensuring that no cycles are formed. To achieve this, it uses a min-heap to prioritize edges and a union-find data structure to manage disjoint sets (connected components).

A min-heap to store all the edges prioritized by their weights

A min-heap is used to prioritize edges based on weight. All m edges are inserted into a heap before the MST construction begins. An array H[1..m] is used to hold edge records with fields i, j, and w, representing an edge between nodes i and j with weight w.

Example: For a graph with n = 64 and m = 512, the heap may contain:

H = [{'i': 3, 'j': 9, 'w': 104}, {'i': 12, 'j': 29, 'w': 37}, ...]

Timing Rule: The construction of the heap (inserting all edges and heapifying) must be included in the timing, but gathering edges from the matrix should not be.

N trees representing n connected components in the beginning

The algorithm starts with n disjoint sets (trees), one for each node. It maintains a forest that merges components as edges are added.

Data Structures:

- parent[1..n]: Keeps track of the root (or parent) of each node. Initially, parent[i] = i.
- rank[1..n]: Stores the rank (approximate height) of each tree. Helps to optimize union operations.

Find Operation (C-Find):

- Returns the representative (root) of a node's set.
- Uses path compression to speed up repeated lookups.
- Example: Find(29) might return 12 if node 29 is part of the tree rooted at node 12.

```
def find(parent, i):
    # Path compression for Union-Find
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]
```

Union Operation (W-Union):

- Merges two sets. Uses rank to attach the smaller tree under the root of the larger one.
- Example: If rank[12] = 3 and rank[29] = 1, then the root of 29 is updated to point to 12.

```
def union(parent, rank, x, y):
    # Union by rank to optimize merging
    xroot = find(parent, x)
    yroot = find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
```

Avoiding Cycles:

- Before adding an edge {i, j}, Kruskal checks if Find(i) != Find(j).
- If true, the edge is safe to add; otherwise, it would form a cycle.

A variable TotalWeight which will accumulate the weight of the edges in the spanning forest as it is formed

Two additional constructs are used to track the MST. One is **TotalWeight**, a scalar that keeps the cumulative sum of the weights of edges added to the MST.

TotalWeight = 0

An array T [1..n - 1] implementing the set of edges in the spanning tree

T[1..n-1]: An array storing the edges included in the MST. This helps verify correctness.

Example: After completion,

T = [(3, 9), (12, 29), ..., (55, 62)]

Prim's Algorithm

Prim's algorithm incrementally grows the MST from a starting node by always connecting the closest non-MST node. It relies on a min-heap for nodes, an adjacency list for fast lookups, and a NEAR array to track nearest connections.

The array NEAR[1..n]

Prim's algorithm maintains an array NEAR[1..n] where:

- NEAR[i] stores the minimum known cost to reach node i from the growing MST.
- It is initialized to ∞ for all nodes except the starting node.

Example for n = 64:

```
NEAR = [0, inf, inf, ..., inf]  # Start from node 0
```

Min-Heap to Store Nodes

Unlike Kruskal's edge-based heap, Prim's heap stores nodes prioritized by their current closest edge cost.

Structure:

```
heap = [(0, 0)]  # (cost, node)
```

Operations:

- Extract-Min: Removes the closest node to the current MST.
- Decrease-Key: Updates a node's cost if a shorter path is found.

Example:

- Node 27 discovered with new cost 12 → heap.push((12, 27)).
- Later, if an edge with weight 9 to node 27 is found → update heap with heap.decrease_key(27, 9).

Adjacency List Representation

Prim's algorithm needs fast access to neighbors, which is achieved via adjacency lists.

Construction:

```
adj = [[] for _ in range(n)]
adj[3].append((9, 104))  # Edge from 3 to 9 with weight 104
adj[9].append((3, 104))
```

Timing Rule: Building the adjacency list is not included in the timing. Only the MST construction operations should be timed.

Usage:

- After extracting node u from the heap, its neighbors are scanned via adj[u].
- Each neighbor v is considered for addition or update in the heap.

Like Kruskal, Prim also uses:

- **TotalWeight**: Sum of selected edge weights.
- T[1..n-1]: Array of MST edges, e.g., [(0, 3), (3, 9), ..., (55, 62)].

Graph Representation (for Both Algorithms)

Both Kruskal and Prim can work off an adjacency list or adjacency matrix:

- Kruskal: Can scan upper/lower triangle of matrix W[i][j] to gather edges.
- Prim: Requires adjacency list for fast neighbor access.

For n = 64, m = 512:

- The maximum number of unique undirected edges is 2016.
- An average degree is 2m/n = 16, implying moderate density.

Edge Scanning for Kruskal:

- Can be done once at the beginning to populate heap.
- Should not be included in timing.