Prim’s algorithm is used to find the Minimum Spanning Tree (MST) of a connected, weighted graph. The algorithm starts with an arbitrary vertex and grows the MST by adding the shortest edge that connects a vertex in the MST to a vertex outside the MST. The algorithm continues until all vertices are included in the MST.

Prim’s Algorithm Pseudocode

1. **Input:**
    - A connected graph G=(V,E)G = (V, E)G=(V,E), where VVV is the set of vertices and EEE is the set of edges, with weights w(u,v)w(u, v)w(u,v) for each edge (u,v)∈E(u, v) \in E(u,v)∈E.
2. **Initialization:**
    - Start with any arbitrary vertex v0∈Vv_0 \in Vv0∈V (you can choose the first vertex arbitrarily).
    - Set the MST as empty.
    - Set up a priority queue (min-heap) to select the edge with the minimum weight.
    - Initialize an array key[]key[]key[] where each key[v]key[v]key[v] is set to infinity except for the starting vertex, which is set to 0.
    - Initialize an array parent[]parent[]parent[] to store the parent of each vertex.
3. **Algorithm Steps:**

1. Insert all vertices into the priority queue, with the key values representing their current smallest edge weight.

2. While the priority queue is not empty:
    - Extract the vertex uuu with the smallest key value from the queue.
    - For each neighbor vvv of uuu:
        - If vvv is not in the MST and the edge weight w(u,v)w(u, v)w(u,v) is smaller than key[v]key[v]key[v]:
            - Update key[v]key[v]key[v] to w(u,v)w(u, v)w(u,v).
            - Update parent[v]parent[v]parent[v] to uuu.
            - Update the priority queue with the new key value of vvv.

3. When the loop ends, the MST is formed. The edges included in the MST can be determined from the parent[] array.

4. **Output:**
    - The MST is the set of edges corresponding to the parent[] array, excluding the starting vertex.

Prim’s Algorithm (Pseudocode)

plaintext

Copy code

```
Prim-MST(Graph G, start vertex)
  Input: A connected weighted graph G(V, E), a starting vertex `start`
  Output: Minimum Spanning Tree (MST)

  Initialize a priority queue (min-heap) Q
  Initialize arrays:
    key[] with ∞ for all vertices except key[start] = 0
    parent[] with null values (no parent at first)
    visited[] with false (to track visited vertices)

  Insert all vertices into the priority queue Q with their key values
  while Q is not empty:
    u = Extract-Min(Q)  // Extract the vertex u with the minimum key
    visited[u] = true   // Mark the vertex u as included in MST

    for each neighbor v of u:
      if visited[v] == false and weight(u, v) < key[v]:
        key[v] = weight(u, v)
        parent[v] = u
        Decrease-Key(Q, v, key[v])

  return parent[]  // Contains the MST edges
```

Part (b): Analyze in Detail Your Written Algorithms

Time Complexity Analysis:

1. **Initialization:**
    - Initializing the key[] and parent[] arrays takes O(V)O(V)O(V), where VVV is the number of vertices.
2. **Priority Queue Operations:**
    - **Insert operation**: Each insertion into the priority queue takes O(log⁡V)O(\log V)O(logV), and since we perform this operation for all VVV vertices, the total complexity for insertion is O(Vlog⁡V)O(V \log V)O(VlogV).
    - **Extract-Min operation**: Each extraction takes O(log⁡V)O(\log V)O(logV), and we perform this operation VVV times, so the total complexity for extractions is O(Vlog⁡V)O(V \log V)O(VlogV).
    - **Decrease-Key operation**: Each edge (u,v)(u, v)(u,v) requires one decrease-key operation if the weight of the edge is smaller than the current value in key[]. Since there are EEE edges, the total complexity for decrease-key operations is O(Elog⁡V)O(E \log V)O(ElogV).
3. **Total Complexity:** The total time complexity of Prim's algorithm is dominated by the priority queue operations, specifically O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where VVV is the number of vertices and EEE is the number of edges in the graph.

So, the **time complexity** of Prim's algorithm is:

O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV)

Space Complexity:

- We need space to store the graph, which has O(V+E)O(V + E)O(V+E) space.
- Arrays for key[], parent[], and visited[] each require O(V)O(V)O(V) space.
- The priority queue requires O(V)O(V)O(V) space.

Thus, the **space complexity** of Prim's algorithm is:

O(V+E)O(V + E)O(V+E)

Characteristics of Prim's Algorithm:

- **Greedy Algorithm**: Prim's algorithm is a greedy algorithm because it chooses the edge with the smallest weight to add to the MST at each step.
- **Efficient for Dense Graphs**: When the number of edges EEE is large, Prim's algorithm can be more efficient than Kruskal's algorithm, as it operates in O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), while Kruskal's algorithm would require sorting the edges first, which has a time complexity of O(Elog⁡E)O(E \log E)O(ElogE).
- **Works with All Types of Graphs**: Prim's algorithm works for both directed and undirected graphs.

Part (c): Implement Your Written Algorithms in Part (a)

Below is the Python implementation of Prim's algorithm to find the Minimum Spanning Tree (MST):

python

Copy code

```
import heapq

def prim_mst(graph, start):
    V = len(graph)  # Number of vertices
    key = [float('inf')] * V  # Initialize keys as infinity
    parent = [None] * V  # Array to store the parent of each vertex
    visited = [False] * V  # Track visited vertices
    key[start] = 0  # Start with the given start vertex

    # Min-heap priority queue
    pq = [(0, start)]  # (key, vertex)

    while pq:
        # Extract vertex with minimum key value
        current_key, u = heapq.heappop(pq)
        visited[u] = True

        # Traverse all adjacent vertices of u
        for v, weight in enumerate(graph[u]):
            if not visited[v] and weight != 0 and weight < key[v]:
                key[v] = weight
                parent[v] = u
                heapq.heappush(pq, (key[v], v))
    # The parent[] array contains the MST edges
    return parent


# Example usage:
# Graph represented as an adjacency matrix where graph[u][v] is the weight of the edge between u and v.
graph = [
    [0, 2, 0, 6, 0],
    [2, 0, 3, 8, 5],
    [0, 3, 0, 0, 7],
    [6, 8, 0, 0, 9],
    [0, 5, 7, 9, 0]
]


start_vertex = 0
parent = prim_mst(graph, start_vertex)


# Print the MST edges by showing the parent array
print("Edge   Weight")
for i in range(1, len(parent)):
    print(f"{parent[i]} - {i}   {graph[i][parent[i]]}")
```

Explanation of the Python Code:

1. **Graph Representation**: The graph is represented as an adjacency matrix where the value at position graph[u][v] holds the weight of the edge between vertices u and v. A value of 0 means there is no edge between the vertices.
2. **Priority Queue (Min-Heap)**: A priority queue (min-heap) is used to extract the vertex with the minimum key value efficiently. Python's heapq module provides the functionality for the min-heap.
3. **Prim's Algorithm Logic**:
    - We initialize the key[] array with infinity and the parent[] array to store the parent of each vertex in the MST.

  - Starting from the given vertex, we update the key[] values for all adjacent vertices and add them to the priority queue.
  - The algorithm ensures that the smallest edge weight is always chosen, and the process continues until all vertices are included in the MST.

Example Output:

For the graph given in the code:

plaintext

Copy code

```
Edge   Weight

0 - 1   2

0 - 3   6

1 - 2   3

1 - 4   5
```

This output shows the edges included in the MST and their corresponding weights.