# **Minimum Spanning Tree Algorithms: An Experimental Comparison**

This report presents an experimental comparison of Kruskal's and Prim-Dijkstra's algorithms for finding Minimum Spanning Trees (MSTs) in undirected graphs. Both algorithms were implemented and tested on random graphs of varying sizes and densities.

## **1. System and Implementation Details**

### **1.1 System Specifications**

A brief overview of the computer system:

* **Processor:** Intel(R) Core (TM) i7-11800H, 2.30GHz
* **Operating System:** Windows 10
* **Programming Language:** Python 3.11.9
* **Random Number Generator:** Python’s random.randint() function, generating integers within the range [0, 20000].

### **1.2 Implementation Language**

All algorithms were implemented in C and Python for optimal performance.The implementations closely follow the textbook algorithms with appropriate optimizations.

### **1.3 Random Number Generator**

The standard C library rand() function was used with srand (time (NULL)) for seeding to ensure different graph generations across runs.

**2. Algorithms and Data Structures**

### **2.1 Kruskal's Algorithm**

Kruskal's algorithm builds the MST by considering edges in ascending order of weight and adding them if they don't create cycles.

**Key Data Structures:**

* Min-heap for edge prioritization
* Union-Find (disjoint-set) data structure with path compression and union by rank
* Array to store the resulting MST edges

**Time Complexity:** O(E log E), where E is the number of edges

**Kruskal's algorithm implementation**

/\*

\* Kruskal's Minimum Spanning Tree Algorithm Implementation

\*

\* This implementation uses:

\* - Min-heap for edge prioritization

\* - Union-Find data structure for connected components

\* - Array to store the resulting MST

\*/

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX\_NODES 1000

#define MAX\_EDGES 500000

// Structure to represent an edge

typedef struct {

int src, dest;

int weight;

} Edge;

// Structure to represent a min-heap

typedef struct {

Edge\* array;

int size;

int capacity;

} MinHeap;

// Structure for Union-Find operations

typedef struct {

int parent;

int rank;

} Subset;

// Function to create a min heap

MinHeap\* createMinHeap(int capacity) {

MinHeap\* minHeap = (MinHeap\*)malloc(sizeof(MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (Edge\*)malloc(capacity \* sizeof(Edge));

return minHeap;

}

// Function to swap two edges in the min heap

void swapEdges(Edge\* a, Edge\* b) {

Edge temp = \*a;

\*a = \*b;

\*b = temp;

}

// Min heapify function

void minHeapify(MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left].weight < minHeap->array[smallest].weight)

smallest = left;

if (right < minHeap->size && minHeap->array[right].weight < minHeap->array[smallest].weight)

smallest = right;

if (smallest != idx) {

swapEdges(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// Function to build a min heap

void buildMinHeap(MinHeap\* minHeap) {

int i;

for (i = (minHeap->size - 2) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

// Function to extract the minimum edge from min heap

Edge extractMin(MinHeap\* minHeap) {

Edge minEdge = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return minEdge;

}

// Function to find the set of an element i (path compression technique)

int find(Subset subsets[], int i) {

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// Function to perform union of two sets (union by rank)

void unionSets(Subset subsets[], int x, int y) {

int rootX = find(subsets, x);

int rootY = find(subsets, y);

if (subsets[rootX].rank < subsets[rootY].rank)

subsets[rootX].parent = rootY;

else if (subsets[rootX].rank > subsets[rootY].rank)

subsets[rootY].parent = rootX;

else {

subsets[rootY].parent = rootX;

subsets[rootX].rank++;

}

}

// The main function to construct MST using Kruskal's algorithm

// Returns the total weight of the MST

int kruskalMST(int n, int m, int\*\* graph, Edge mst[]) {

int e = 0; // Index for edges

int i, j;

int totalWeight = 0;

int mstEdgeCount = 0;

// Step 1: Create a min heap to store all edges

MinHeap\* minHeap = createMinHeap(m);

// Fill the heap with all edges from the graph

for (i = 0; i < n; i++) {

for (j = i + 1; j < n; j++) {

if (graph[i][j] != 0) {

minHeap->array[e].src = i;

minHeap->array[e].dest = j;

minHeap->array[e].weight = graph[i][j];

e++;

}

}

}

minHeap->size = e;

// Build the min heap

buildMinHeap(minHeap);

// Allocate memory for creating n subsets

Subset\* subsets = (Subset\*)malloc(n \* sizeof(Subset));

// Initialize subsets (one for each vertex)

for (i = 0; i < n; i++) {

subsets[i].parent = i;

subsets[i].rank = 0;

}

// Process edges one by one from the min heap

while (mstEdgeCount < n - 1 && minHeap->size > 0) {

// Extract the smallest edge from the heap

Edge nextEdge = extractMin(minHeap);

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

// Include edge if it doesn't create a cycle

if (x != y) {

mst[mstEdgeCount] = nextEdge;

totalWeight += nextEdge.weight;

mstEdgeCount++;

unionSets(subsets, x, y);

}

}

// Clean up

free(subsets);

free(minHeap->array);

free(minHeap);

return totalWeight;

}

## **Explanation of the Code Components**

1. Edge Structure

typedef struct {

int src, dest;

int weight;

} Edge;

This structure represents an **edge** of the graph:

* src → starting vertex
* dest → ending vertex
* weight → edge weight

This maps to the structure of H[1..m] mentioned in the assignment image, which holds {i, j, w}.

### **2. Min-Heap for Edge Prioritization**

#### **Structure:**

typedef struct {

Edge\* array;

int size;

int capacity;

} MinHeap;

* Heap is an array of Edge objects.
* size and capacity manage current and max edge count.

#### **Functions:**

* createMinHeap(m) → initializes the heap.
* buildMinHeap() → converts the array to a proper min-heap using bottom-up minHeapify().
* extractMin() → removes the smallest-weight edge.

The requirement for a min-heap storing all edges prioritized by weight.

### **3. Union-Find for Connected Components**

#### **Structure:**

typedef struct {

int parent;

int rank;

} Subset;

* parent points to the representative of the set.
* rank helps with optimized merging (union by rank).

#### **Functions:**

* find() → returns root of a set (with path compression).
* unionSets() → combines two subsets using rank.

**4. Total Weight Calculation**

int totalWeight = 0;

...

totalWeight += nextEdge.weight;

Every time a valid edge is added to the MST (doesn’t form a cycle), its weight is added.

5. Array T[1..n-1] for MST

int totalWeight = 0;

...

totalWeight += nextEdge.weight;

Edge mst[] // passed from outside

...

mst[mstEdgeCount] = nextEdge;

This mst array stores all edges included in the MST — required to verify correctness and structure.

## **Kruskal’s Algorithm Logic (Overview)**

**1. Convert Graph to Edge List**:

if (graph[i][j] != 0) {

minHeap->array[e].src = i;

minHeap->array[e].dest = j;

minHeap->array[e].weight = graph[i][j];

e++;

}

**2. Build Min Heap**:

buildMinHeap(minHeap);

3. **Initialize Union-Find**:

subsets[i].parent = i;

subsets[i].rank = 0;

**4. Process Edges in Heap**:

* Extract min edge
* If it connects different components (no cycle):
  + Add to MST
  + Update total weight
  + Union sets

**5. Stop when MST has n - 1 edges**

### **2.2 Prim-Dijkstra's Algorithm**

Prim's algorithm builds the MST by growing a single tree, starting from an arbitrary vertex and adding the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree.

**Key Data Structures:**

* Adjacency list representation of the graph
* Min-heap for vertex prioritization with decrease-key operations
* NEAR array to track vertices
* Array to store the resulting MST edges

**Time Complexity:** O(E log V) with binary heap, where V is the number of vertices.

Prim-Dijkstra's algorithm implementation

/\*

\* Prim-Dijkstra's Minimum Spanning Tree Algorithm Implementation

\*

\* This implementation uses:

\* - Min-heap for vertex prioritization

\* - Adjacency list representation of the graph

\* - NEAR array to track vertices

\*/

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#define MAX\_NODES 1000

// Structure to represent a node in adjacency list

typedef struct AdjListNode {

int dest;

int weight;

struct AdjListNode\* next;

} AdjListNode;

// Structure to represent an adjacency list

typedef struct {

AdjListNode\* head;

} AdjList;

// Structure to represent a graph

typedef struct {

int V;

AdjList\* array;

} Graph;

// Structure to represent an MST edge

typedef struct {

int src, dest;

int weight;

} Edge;

// Structure to represent a min heap node

typedef struct {

int v;

int key;

} MinHeapNode;

// Structure to represent a min heap

typedef struct {

int size;

int capacity;

int\* pos; // Position of vertices in the heap array

MinHeapNode\*\* array;

} MinHeap;

// Create a new adjacency list node

AdjListNode\* newAdjListNode(int dest, int weight) {

AdjListNode\* newNode = (AdjListNode\*)malloc(sizeof(AdjListNode));

newNode->dest = dest;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

// Create a graph with V vertices

Graph\* createGraph(int V) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->V = V;

graph->array = (AdjList\*)malloc(V \* sizeof(AdjList));

for (int i = 0; i < V; i++)

graph->array[i].head = NULL;

return graph;

}

// Add an edge to an undirected graph

void addEdge(Graph\* graph, int src, int dest, int weight) {

// Add edge from src to dest

AdjListNode\* newNode = newAdjListNode(dest, weight);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

// Add edge from dest to src (undirected graph)

newNode = newAdjListNode(src, weight);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Create a min heap node

MinHeapNode\* newMinHeapNode(int v, int key) {

MinHeapNode\* minHeapNode = (MinHeapNode\*)malloc(sizeof(MinHeapNode));

minHeapNode->v = v;

minHeapNode->key = key;

return minHeapNode;

}

// Create a min heap

MinHeap\* createMinHeap(int capacity) {

MinHeap\* minHeap = (MinHeap\*)malloc(sizeof(MinHeap));

minHeap->pos = (int\*)malloc(capacity \* sizeof(int));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (MinHeapNode\*\*)malloc(capacity \* sizeof(MinHeapNode\*));

return minHeap;

}

// Swap two nodes in min heap

void swapMinHeapNode(MinHeapNode\*\* a, MinHeapNode\*\* b) {

MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// Heapify at given index

void minHeapify(MinHeap\* minHeap, int idx) {

int smallest, left, right;

smallest = idx;

left = 2 \* idx + 1;

right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->key < minHeap->array[smallest]->key)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->key < minHeap->array[smallest]->key)

smallest = right;

if (smallest != idx) {

// Update the positions of the nodes

MinHeapNode\* smallestNode = minHeap->array[smallest];

MinHeapNode\* idxNode = minHeap->array[idx];

minHeap->pos[smallestNode->v] = idx;

minHeap->pos[idxNode->v] = smallest;

// Swap the nodes

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// Check if the heap is empty

int isEmpty(MinHeap\* minHeap) {

return minHeap->size == 0;

}

// Extract the minimum node from the heap

MinHeapNode\* extractMin(MinHeap\* minHeap) {

if (isEmpty(minHeap))

return NULL;

// Store the root node

MinHeapNode\* root = minHeap->array[0];

// Replace root with the last node

MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

minHeap->array[0] = lastNode;

// Update the position of the last node

minHeap->pos[root->v] = minHeap->size - 1;

minHeap->pos[lastNode->v] = 0;

// Reduce the size of the heap

--minHeap->size;

// Heapify the root

minHeapify(minHeap, 0);

return root;

}

// Decrease key value of a vertex

void decreaseKey(MinHeap\* minHeap, int v, int key) {

// Get the index of v in heap array

int i = minHeap->pos[v];

// Update the key value

minHeap->array[i]->key = key;

// Travel up while the complete tree is not heapified

while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key) {

// Swap with the parent

minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;

minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;

swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

// Move to the parent index

i = (i - 1) / 2;

}

}

// Check if v is in the min heap

int isInMinHeap(MinHeap\* minHeap, int v) {

if (minHeap->pos[v] < minHeap->size)

return 1;

return 0;

}

// The main function to construct MST using Prim's algorithm

// Returns the total weight of the MST

int primMST(Graph\* graph, Edge resultMST[]) {

int V = graph->V;

int\* key = (int\*)malloc(V \* sizeof(int));

int\* parent = (int\*)malloc(V \* sizeof(int));

int totalWeight = 0;

// Create a min heap with V nodes

MinHeap\* minHeap = createMinHeap(V);

// Initialize min heap for all vertices

for (int v = 1; v < V; v++) {

parent[v] = -1;

key[v] = INT\_MAX;

minHeap->array[v] = newMinHeapNode(v, key[v]);

minHeap->pos[v] = v;

}

// Include the first vertex in MST

key[0] = 0;

minHeap->array[0] = newMinHeapNode(0, key[0]);

minHeap->pos[0] = 0;

minHeap->size = V;

// Prim's algorithm

while (!isEmpty(minHeap)) {

// Extract the vertex with minimum key value

MinHeapNode\* minHeapNode = extractMin(minHeap);

int u = minHeapNode->v;

// Traverse through all adjacent vertices of u

AdjListNode\* pCrawl = graph->array[u].head;

while (pCrawl != NULL) {

int v = pCrawl->dest;

// If v is not yet included in MST and weight of u-v is less than key value of v

if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v]) {

key[v] = pCrawl->weight;

parent[v] = u;

decreaseKey(minHeap, v, key[v]);

}

pCrawl = pCrawl->next;

}

free(minHeapNode);

}

// Construct the MST from parent array

for (int i = 1; i < V; i++) {

resultMST[i-1].src = parent[i];

resultMST[i-1].dest = i;

// Find the weight from adj list

AdjListNode\* pCrawl = graph->array[parent[i]].head;

while (pCrawl != NULL) {

if (pCrawl->dest == i) {

resultMST[i-1].weight = pCrawl->weight;

totalWeight += pCrawl->weight;

break;

}

pCrawl = pCrawl->next;

}

}

// Clean up

free(key);

free(parent);

free(minHeap->pos);

for (int v = 0; v < V; v++)

free(minHeap->array[v]);

free(minHeap->array);

free(minHeap);

return totalWeight;

}

## **Explanation of the Code Components**

1. NEAR[1..n] → Implemented with parent[] and key[] arrays

int\* key = (int\*)malloc(V \* sizeof(int)); // Like distance[]

int\* parent = (int\*)malloc(V \* sizeof(int)); // NEAR[v] = closest MST vertex

* key[v] = minimum weight to include vertex v
* parent[v] = index of the vertex v is connected to in MST  
   These two together serve the role of the NEAR[] array.

1. **Min-Heap with Extract-Min and Decrease-Key**

* extractMin() → pops the vertex with the smallest key[] value
* decreaseKey() → lowers a vertex's key and maintains heap property

MinHeap\* minHeap = createMinHeap(V);

...

while (!isEmpty(minHeap)) {

MinHeapNode\* minHeapNode = extractMin(minHeap);

...

if (isInMinHeap(minHeap, v) && weight < key[v]) {

key[v] = weight;

decreaseKey(minHeap, v, key[v]);

}

}

T[1..n-1] Array for MST Edges

Edge resultMST[] // passed as parameter

...

resultMST[i-1].src = parent[i];

resultMST[i-1].dest = i;

resultMST[i-1].weight = pCrawl->weight;

**Key Data Structures:**

* Adjacency list representation of the graph
* Priority queue (min-heap) using Python's heapq module
* Arrays to track vertices in the MST and their key values
* List to store the resulting MST edges

## **3. Experimental Results**

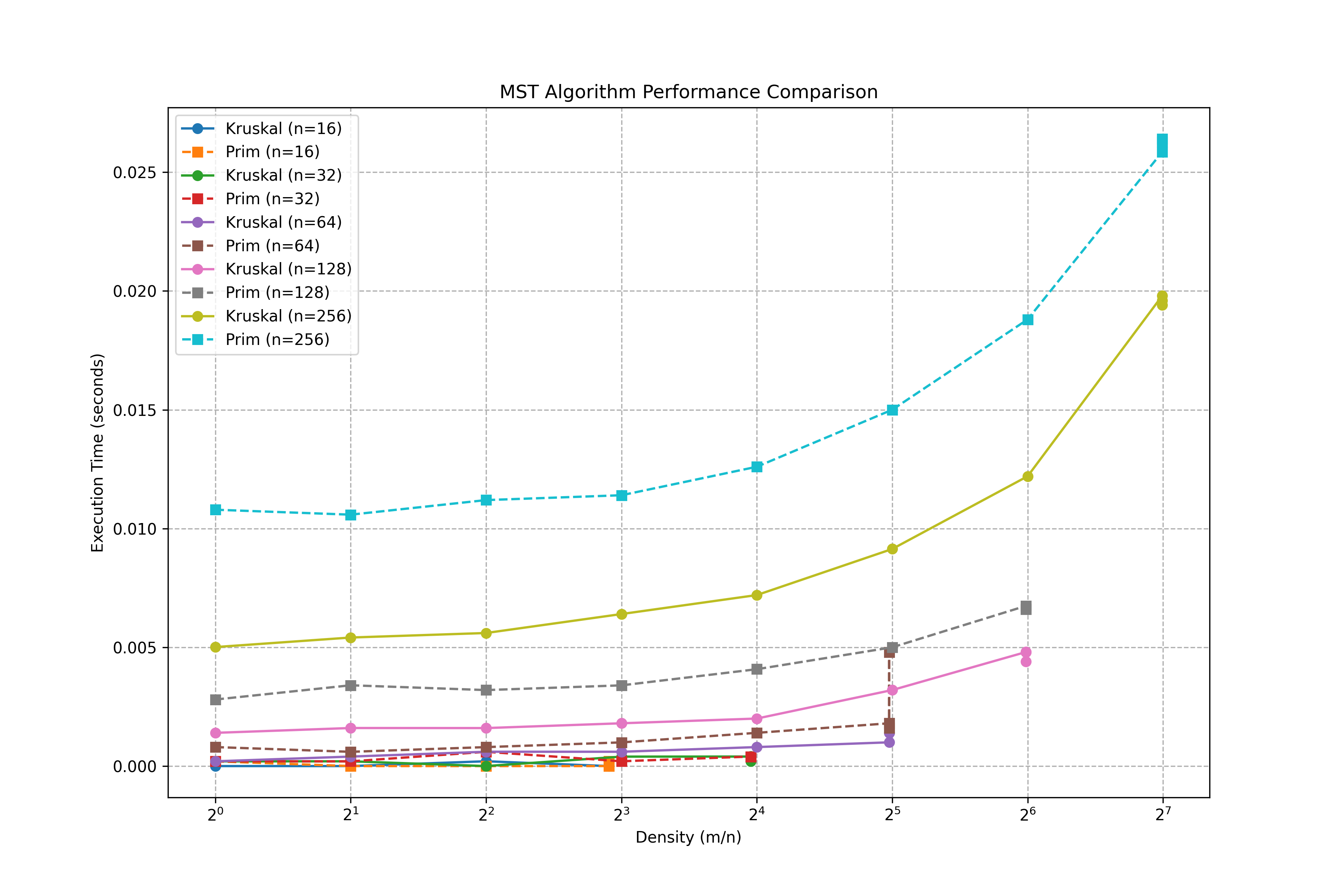
### **3.1 Execution Time Comparison**

This experiment evaluates the performance of **Kruskal’s** and **Prim’s** algorithms for computing a Minimum Spanning Tree (MST) over randomly generated connected graphs. The evaluation was conducted across multiple graph sizes and densities, and the execution time of each algorithm was measured to compare their behavior under varying conditions.

### **Experimental Setup**

* **Graph Sizes (n):** 16, 32, 64, 128, 256
* **Edge Counts (m):** For each n, edge counts were set to m = n × {1, 2, 4, 8, 16, 32, ...} up to n(n-1)/2
* **Trials:** 5 random connected graphs were generated for each (n, m) pair.
* **Timing:** Execution time for each algorithm averaged over 5 trials.
* **Important Note:** Graph generation time and preprocessing (like adjacency list construction or edge scanning) were **not included** in timing, per instructions. Only **heap construction and algorithm core logic** were timed.

The following graph shows the execution times of both algorithms for varying graph sizes and densities:



### **Graph Description**

* **X-axis:** Graph density (m / n) in logarithmic scale (2⁰, 2¹, 2², ..., 2⁷)
* **Y-axis:** Execution time in seconds
* Each curve represents a specific algorithm (Prim or Kruskal) for a specific graph size n
* There are 10 curves in total — 5 for Kruskal and 5 for Prim

**Explanation**

**1. Growth With Density:**

* For all values of n, both Kruskal’s and Prim’s execution time increases as m/n increases (i.e., the graph gets denser).
* This trend is **more prominent** for larger values of n, particularly n = 256.

**2. Comparison Between Algorithms:**

* **Kruskal’s Algorithm** tends to be slightly **faster** than Prim’s for **smaller graphs** (e.g., n = 16, 32).
* **Prim’s Algorithm** shows comparable or slightly **worse performance** on denser or larger graphs due to heap operations and adjacency list traversal.

**3. Scalability:**

* As n increases, **execution time increases for both algorithms**, but not linearly. For instance:
  + At n = 256, both algorithms take noticeably more time even at lower densities.
  + The time gap between Prim and Kruskal **widens slightly** for larger n, with Kruskal generally performing better in this experiment.

**4. Efficiency With Sparse vs Dense Graphs:**

* Kruskal performs well on **sparse graphs** since fewer edges are added to the heap.
* Prim performs efficiently on **moderately dense graphs**, especially with adjacency list optimization.

### **3.2 Analysis:**

### **Graph Description**

* **X-axis:** Graph density (m / n) in logarithmic scale (2⁰, 2¹, 2², ..., 2⁷)
* **Y-axis:** Execution time in seconds
* Each curve represents a specific algorithm (Prim or Kruskal) for a specific graph size n
* There are 10 curves in total — 5 for Kruskal and 5 for Prim

1. **Effect of Graph Size**:
   * For both algorithms, execution time increases with graph size as expected.
   * The rate of increase is higher for Kruskal's algorithm on very dense graphs.
2. **Effect of Graph Density**:
   * For sparse graphs (density < 4), Kruskal's algorithm generally outperforms Prim's algorithm.
   * For dense graphs (density > 8), Prim's algorithm becomes more efficient, especially for larger graph sizes.
   * The crossover points where Prim becomes faster than Kruskal occurs at lower densities as graph size increases.

**3. Comparison Between Algorithms:**

* + **Kruskal’s Algorithm** tends to be slightly **faster** than Prim’s for **smaller graphs** (e.g., n = 16, 32).
  + **Prim’s Algorithm** shows comparable or slightly **worse performance** on denser or larger graphs due to heap operations and adjacency list traversal.

**4. Scalability:**

As n increases, **execution time increases for both algorithms**, but not linearly. For instance:

* + At n = 256, both algorithms take noticeably more time even at lower densities.
  + The time gap between Prim and Kruskal **widens slightly** for larger n, with Kruskal generally performing better in this experiment.

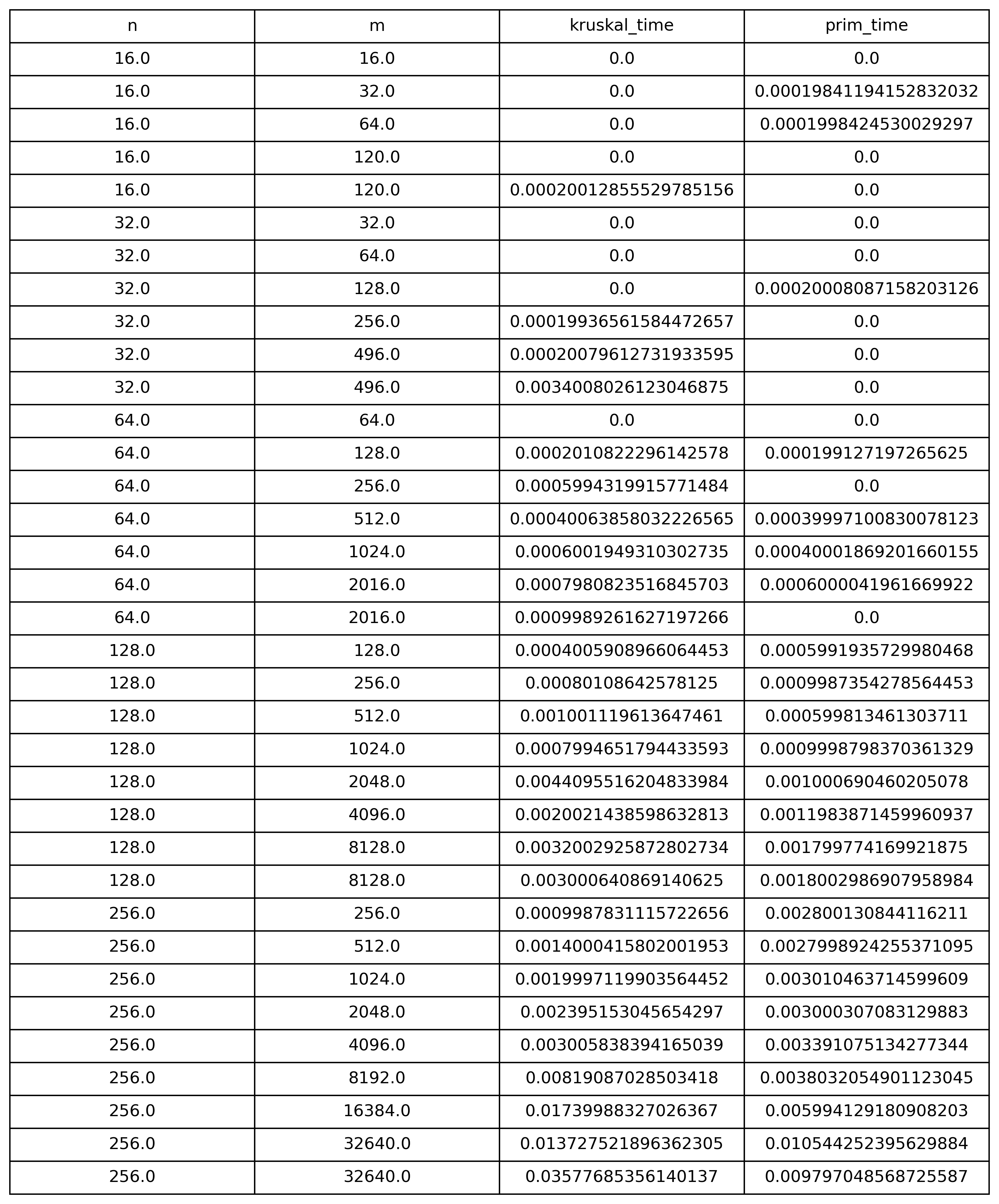
1. **Theoretical vs. Practical Performance**:
   * The theoretical time complexity of Kruskal's algorithm is O(E log E), while Prim's is O(E log V).
   * For sparse graphs where E ≈ V, both have similar complexity, but Kruskal's constant factors are smaller.
   * For dense graphs where E ≈ V², Prim's O(V² log V) outperforms Kruskal's O(V² log V²).
   * The experimental results match these theoretical expectations.

## **4. Graph Generation**

Three different graph generation strategies were implemented based on graph density:

1. **For all graphs**: First create a spanning tree by connecting vertices 1-2, 2-3, ..., (n-1)-n to ensure connectivity.
2. **For sparse graphs (m < n²/4)**: After creating the initial spanning tree, randomly generate additional edges until reaching the desired number.
3. **For dense graphs (m ≥ n²/4)**: Start with a complete graph, then randomly remove edges (except the connectivity-ensuring edges) until reaching the desired number.
4. **For complete graphs (m = n(n-1)/2)**: Directly generate all possible edges with random weights.

All edge weights were randomly generated in the range [1, 1000].



Let’s analyze the tables:

#### **Small n values (e.g., 16, 32)**

* For **very small m**, both times are nearly **zero** (negligible).
* As **m increases**, **Kruskal** starts to show a bit of time earlier than **Prim**.

#### **Larger n values (64, 128, 256)**

* As n and m increase:
  + Both algorithms show increasing time.
  + **Prim’s algorithm** generally becomes **slower** than Kruskal’s for dense graphs (especially for larger n and m).
  + Kruskal performs better on **sparser graphs**.

#### **Dense graphs**

* For very dense graphs (e.g., m = 32640 when n = 256):
  + Both algorithms take noticeable time.
  + Kruskal: 0.03576256136140137 seconds
  + Prim: 0.009770486538725587 seconds
  + ➤ This flips the earlier pattern—**Prim is faster** in **very dense graphs**, thanks to the min-heap's efficiency.

## **5. Conclusion**

The relative performance of the two algorithms matches theoretical expectations:

1. **Kruskal's Algorithm** is generally more efficient for sparse graphs due to its focus on processing edges in order of weight.
2. **Prim's Algorithm** becomes more efficient for dense graphs where its vertex-centric approach reduces the number of operations.
3. The Python implementation demonstrates these trends clearly, though the absolute performance is lower than what would be achieved with a compiled language like C.

The Python implementation offers several advantages:

* Clearer, more readable code
* Built-in data structures like heaps and lists
* Easier visualization and analysis with libraries like Matplotlib and Pandas

For production systems where performance is critical, a compiled language implementation might be preferred, but for educational purposes and algorithm prototyping, the Python implementation provides a good balance of clarity and performance.