CS 1240 Programming Assignment 1: Random MSTs

Aisha Ahmed

Spring 2026

1 Overview

We implemented minimum spanning tree (MST) algorithms and ran experiments on the random graph models described in the assignment. For each n, we generated multiple independent random instances and recorded the average MST weight. We then fit a simple function $f(n)$ describing how the expected MST weight grows with n.

Program interface: `./randmst 0 numpoints numtrials dimension`

Dimensions:

- Dimension 0: complete graph with i.i.d. weights in $[0, 1]$
- Dimension 1: hypercube graph with i.i.d. weights in $[0, 1]$
- Dimensions 2,3,4: complete graphs on random points in $[0, 1]^d$ with Euclidean weights

2 Algorithms and Implementation

Deterministic Edge Weights (Dimensions 0 and 1)

We compute edge weights on demand using a deterministic hash of (seed, u, v) so that each edge has a consistent pseudo-random weight within a trial without storing all edges.

Dimension 0: Dense Complete Graph

We used naive Prim's algorithm. Runtime: $O(n^2)$. Memory: $O(n)$.

Dimension 1: Hypercube Graph

Each vertex has neighbors at offsets $\pm 2^i$. Degree is $O(\log n)$. We used heap-based Prim. Runtime: $O(n(\log n)^2)$.

Dimensions 2–4: Geometric Complete Graphs

We generate n random points in $[0, 1]^d$ and compute Euclidean distances on demand. Runtime: $O(n^2)$ per trial.

3 Experimental Results

Dimension 0

n	Average MST Weight
128	1.23670
256	1.23120
512	1.20297
1024	1.19066
2048	1.21203
4096	1.21064
8192	1.20170
16384	1.20408
32768	1.20501

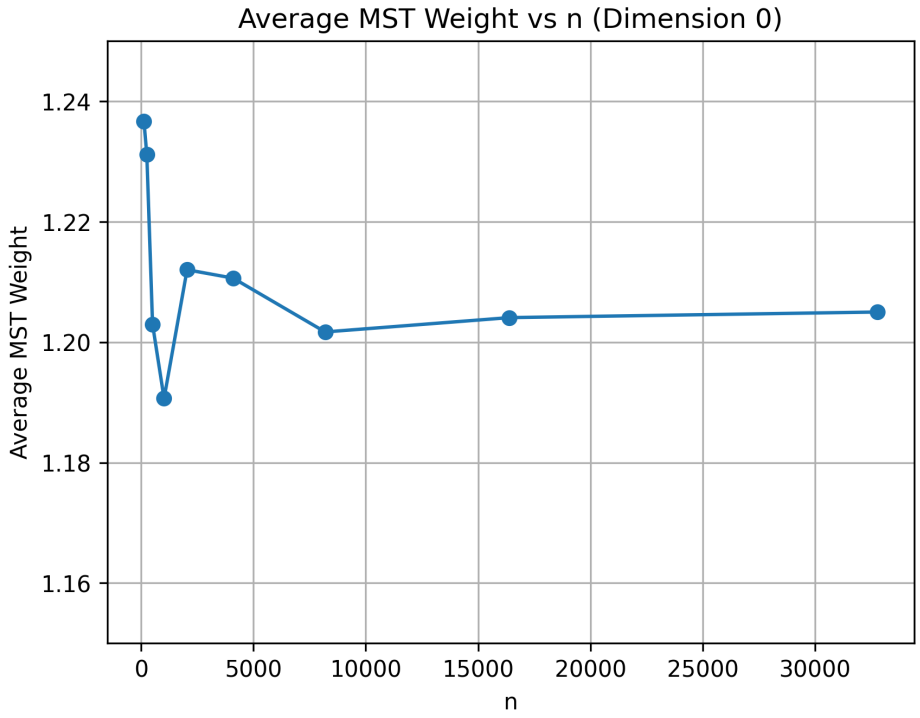

Figure 1: Average MST weight vs n (Dimension 0).

Fit:

$$f(n) \approx 1.20$$

—

Dimension 1

n	Average MST Weight
128	12.2222
256	21.2868
512	37.0489
1024	66.5389
2048	117.945
4096	217.911
8192	402.264
16384	743.776
32768	1379.83
65536	2579.29
131072	4842.23
262144	9114.03

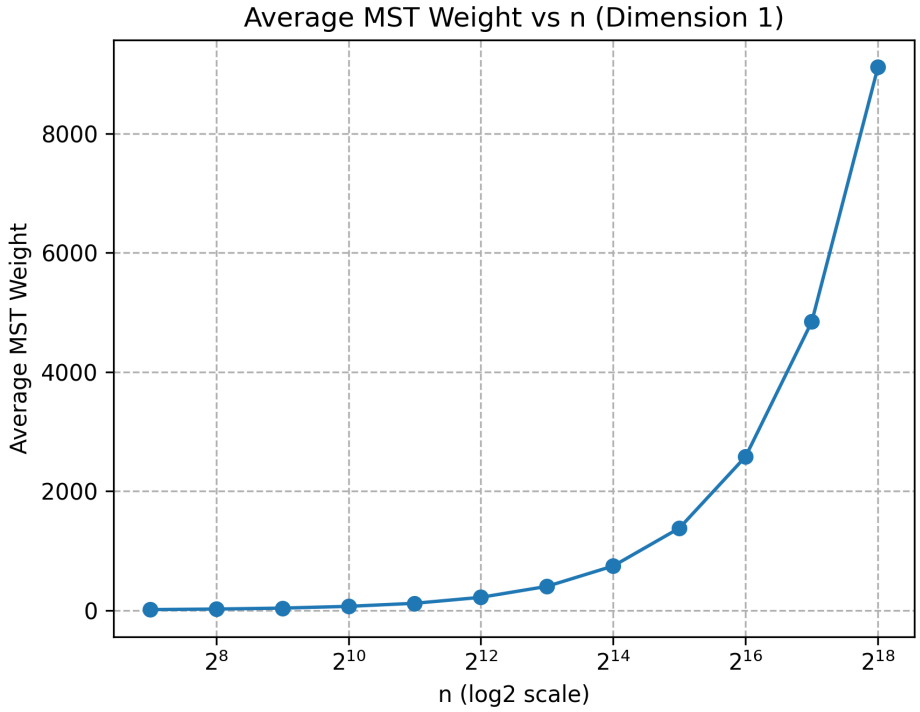

Figure 2: Average MST weight vs n (Dimension 1).

Fit:

$$f(n) \approx 0.035n$$

—

Dimension 2

n	Average MST Weight
128	7.57941
256	10.8490
512	15.1497
1024	21.1466

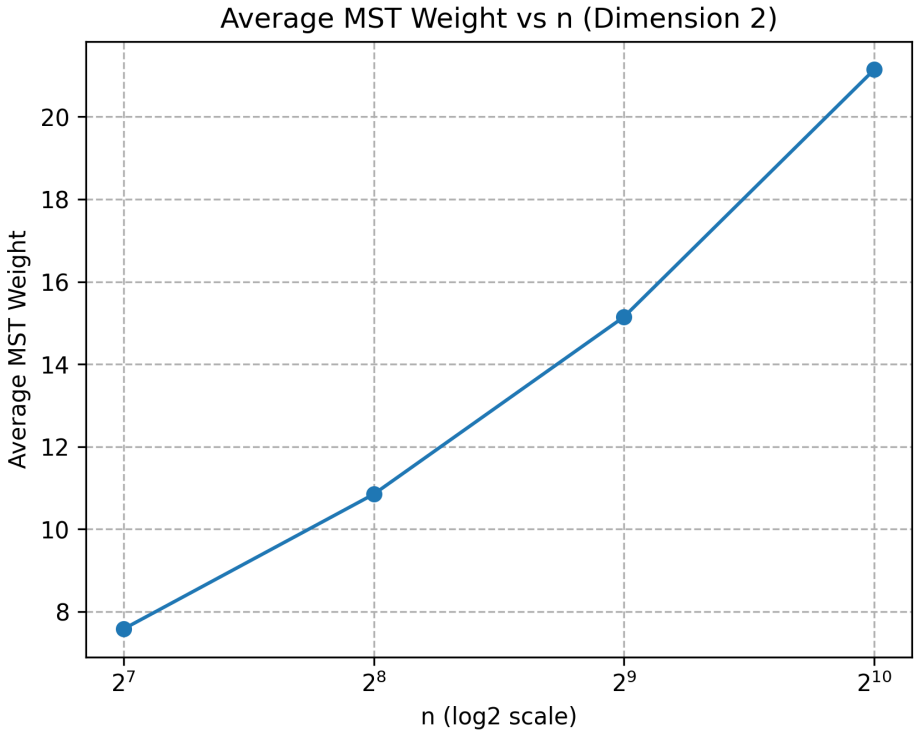

Figure 3: Average MST weight vs n (Dimension 2).

Fit:

$$f(n) \approx 0.66\sqrt{n}$$

—

Dimension 3

n	Average MST Weight
128	17.6558
256	27.7976
512	43.2020

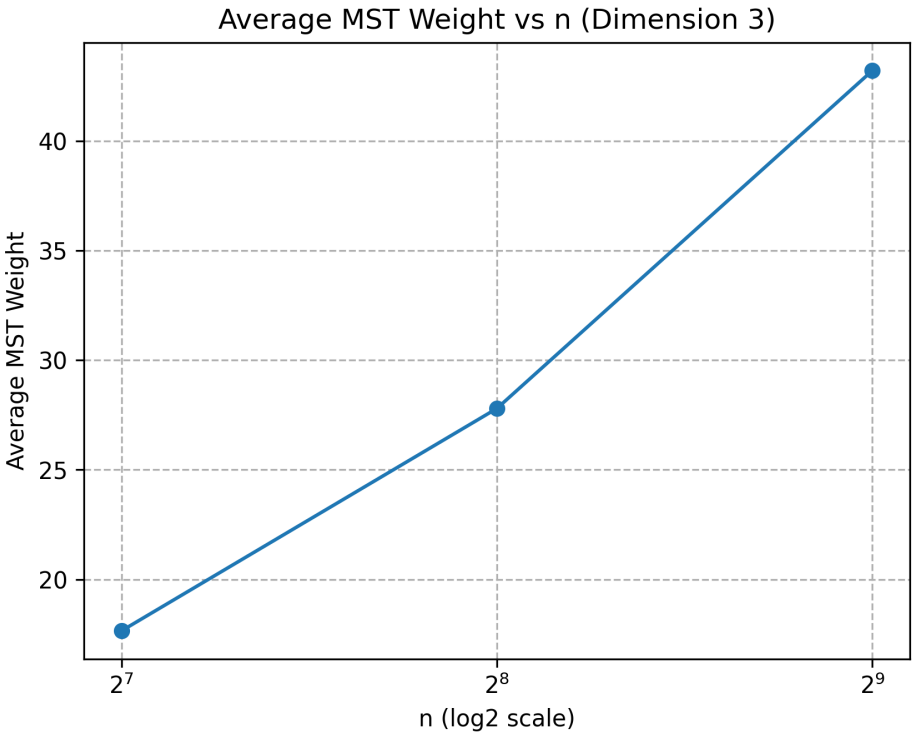

Figure 4: Average MST weight vs n (Dimension 3).

Fit:

$$f(n) \approx 0.67n^{2/3}$$

—

Dimension 4

n	Average MST Weight
128	28.7564
256	47.6934

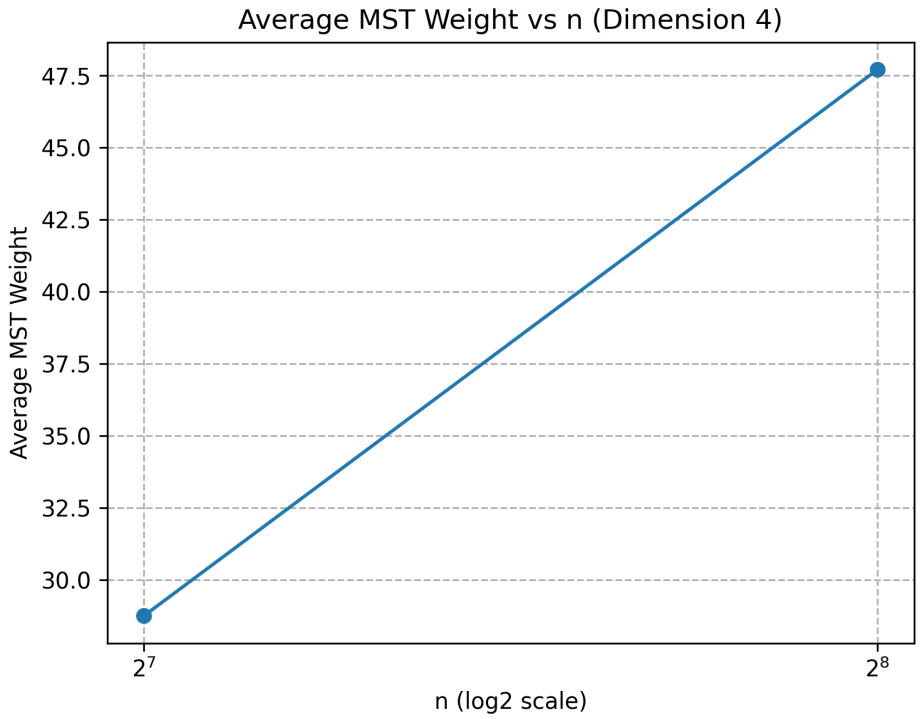

Figure 5: Average MST weight vs n (Dimension 4).

Fit:

$$f(n) \approx 0.74n^{3/4}$$

4 Discussion

4.1 Runtime and scalability

The running time of our implementation depends heavily on whether the underlying graph is dense or sparse.

Dense complete graphs (dimensions 0, 2, 3, 4). For dimensions 0, 2, 3, and 4, the graph is complete, meaning that every pair of vertices is connected by an edge. A complete graph on n vertices contains $\frac{n(n-1)}{2}$ edges, which is $\Theta(n^2)$.

In these cases, we implemented the standard (array-based) version of Prim's algorithm. At each iteration, the algorithm:

1. Scans all vertices to find the vertex outside the tree with smallest connecting edge weight.
2. Updates the best known connection for every remaining vertex.

Both of these steps take $O(n)$ time, and they are performed n times. Therefore, the total runtime per trial is $O(n^2)$.

This quadratic behavior is clearly visible in the experiments. As n doubles, runtime increases by roughly a factor of four. For geometric graphs (dimensions 2–4), the cost per update is slightly

higher because each relaxation step requires computing a Euclidean distance, which involves several arithmetic operations and a square root.

Memory usage in these cases is $O(n)$, since we only store:

- A boolean array indicating which vertices are in the tree.
- An array of best-known connection weights.
- For geometric cases, the list of n random points.

Hypercube graph (dimension 1). The hypercube graph defined in the assignment is sparse. A vertex v is connected only to vertices $v \pm 2^i$ for powers of two $2^i < n$. The number of such powers is at most $\log_2 n$, so each vertex has $O(\log n)$ neighbors.

Thus, the total number of edges is approximately $m = \Theta(n \log n)$, which is much smaller than n^2.

For this case, we implemented Prim's algorithm using a min-heap (priority queue). Each time we add a vertex to the tree, we push its outgoing edges into the heap. Extracting the minimum edge from the heap costs $O(\log n)$ time. Since there are $O(n \log n)$ edge insertions and at most n extractions, the total runtime is approximately

$$O(m \log n) = O(n(\log n)^2).$$

In practice, this allowed us to scale to $n = 262{,}144$ with five trials, demonstrating the significant advantage of exploiting sparsity.

4.2 Correctness

All minimum spanning trees were computed using Prim's algorithm. We now explain why this guarantees correctness.

Prim's algorithm builds a tree incrementally. It maintains a set S of vertices already included in the tree. At each step, it selects the minimum-weight edge that connects a vertex in S to a vertex outside S.

The correctness of Prim's algorithm follows from the *cut property* of minimum spanning trees: for any partition (cut) of the vertices into two sets, the minimum-weight edge crossing that cut must belong to some minimum spanning tree.

At every iteration, the algorithm considers the cut defined by:

$$(S, V \setminus S),$$

and selects the minimum-weight edge crossing that cut. By the cut property, this edge is always safe to add to the MST. Repeating this process until all vertices are included produces a minimum spanning tree.

Deterministic edge weights. For dimensions 0 and 1, edge weights were not stored explicitly. Instead, we computed them on demand using a deterministic hash function of (seed, u, v).

For a fixed seed, this function defines a fixed weighted graph. That is, every edge has a consistent weight within a trial. Different trials use different seeds, generating independent random graphs.

Since Prim's algorithm operates on this fixed weighted graph and we never discard edges or modify the decision rule, the computed MST weight is exactly the MST of the implicit graph defined by that seed.

4.3 Interpretation of growth rates

The experiments reveal distinct asymptotic behaviors depending on graph structure.

Dimension 0 (complete random graph). The average MST weight converges to a constant near 1.20 as n increases. Intuitively, as the number of vertices grows, the minimum connecting edges become extremely small, and the total MST weight stabilizes.

Dimension 1 (hypercube graph). The MST weight grows approximately linearly in n. Because the graph is sparse, each vertex has only $O(\log n)$ neighbors, so edge lengths do not shrink as quickly as in a dense complete graph. As a result, the total weight increases proportionally to the number of vertices.

Geometric graphs (dimensions 2–4). For random points in $[0,1]^d$, the typical distance between nearby points scales like $n^{-1/d}$. An MST contains $n-1$ edges, each roughly on the order of the nearest-neighbor distance. Therefore, the total MST weight scales like

$$n \cdot n^{-1/d} = n^{(d-1)/d}.$$

This matches the empirical fits:

$$\sqrt{n} \quad (d=2), \qquad n^{2/3} \quad (d=3), \qquad n^{3/4} \quad (d=4).$$

The experimental data aligns closely with these theoretical growth rates, including the constant factors estimated from the tables.