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	trials	average MST weight
128	5	1.24752
256	5	1.26799
512	5	1.19621
1024	5	1.22654
2048	5	1.21412
4096	5	1.21436
8192	5	1.19665
16384	5	1.20653
32768	5	1.20153

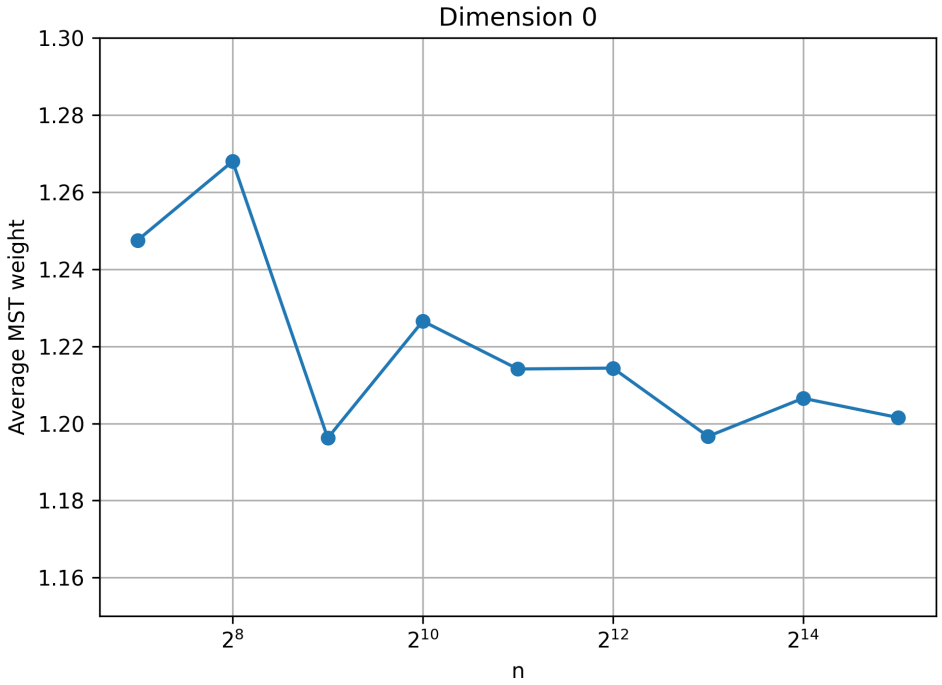

Figure 1: Average MST weight vs. n (Dimension 0).

Fit: The average appears to converge to a constant near 1.20:

$$f(n) \approx 1.20.$$

= =

Dimension 1

n	trials	average MST weight
128	5	12.2464
256	5	21.7467
512	5	36.6067
1024	5	66.4904
2048	5	118.889
4096	5	216.498
8192	5	398.539
16384	5	741.813
32768	5	1377.83
65536	5	2567.20
131072	5	4824.47
262144	5	9098.07

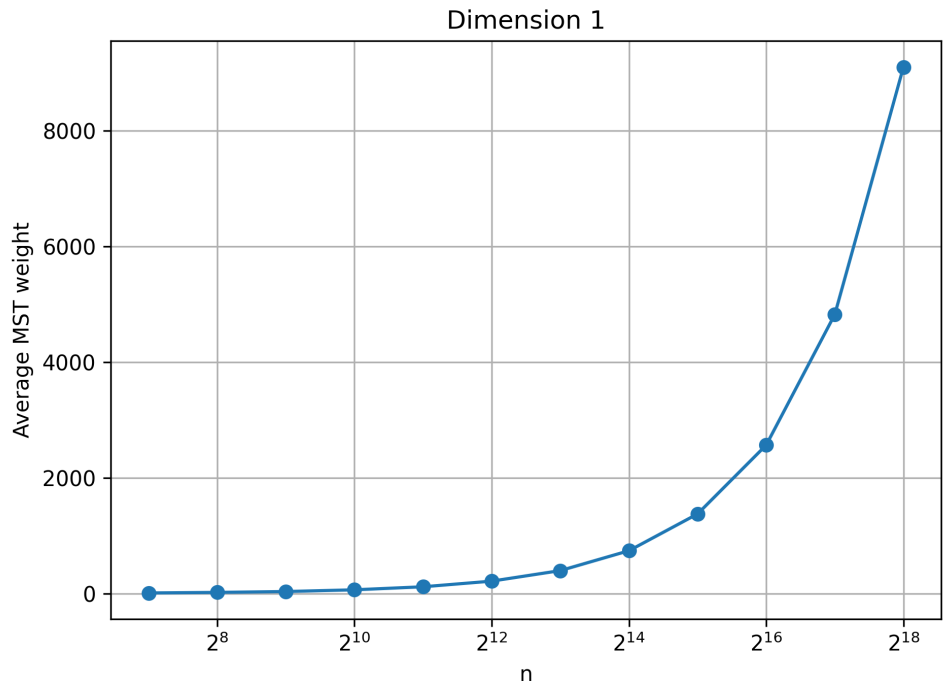

Figure 2: Average MST weight vs. n (Dimension 1).

Fit: Using the largest value,

$$\frac{9098.07}{262144} \approx 0.0347, \qquad f(n) \approx 0.035n.$$

Dimension 2

n	trials	average MST weight
128	5	7.51796
256	5	10.6701
512	5	14.9526
1024	5	21.0870
2048	5	29.6635
4096	5	41.7816

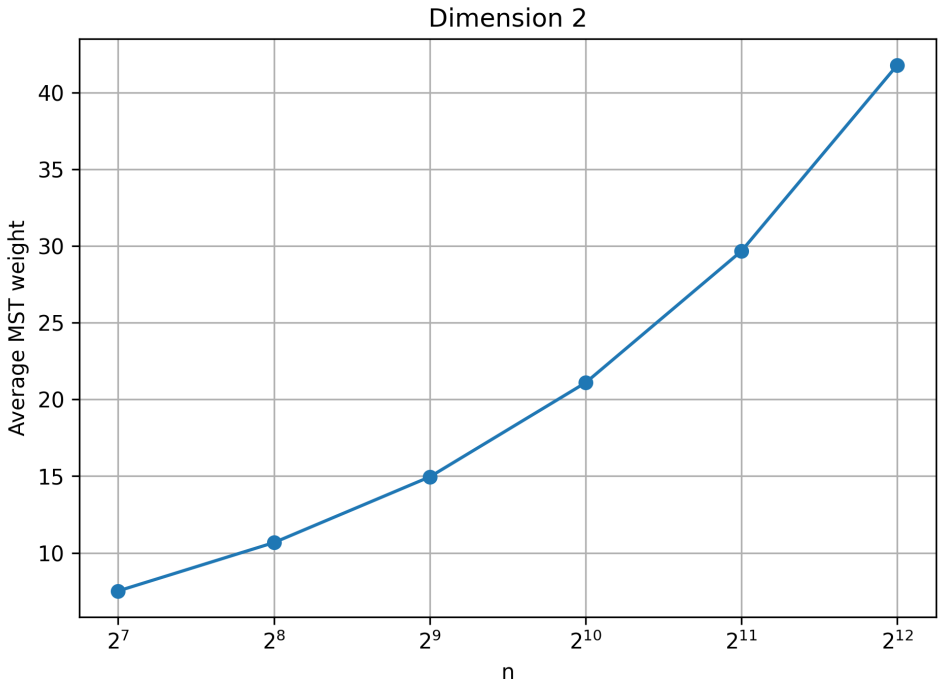

Figure 3: Average MST weight vs. n (Dimension 2).

Fit:
Using $n = 1024$ where $\sqrt{1024} = 32$:

$$c \approx \frac{21.0870}{32} \approx 0.659, \qquad f(n) \approx 0.66\sqrt{n}.$$

Dimension 3

n	trials	average MST weight
128	5	17.7555
256	5	27.4295
512	5	43.3522
1024	5	67.9960
2048	5	106.828
4096	5	169.223

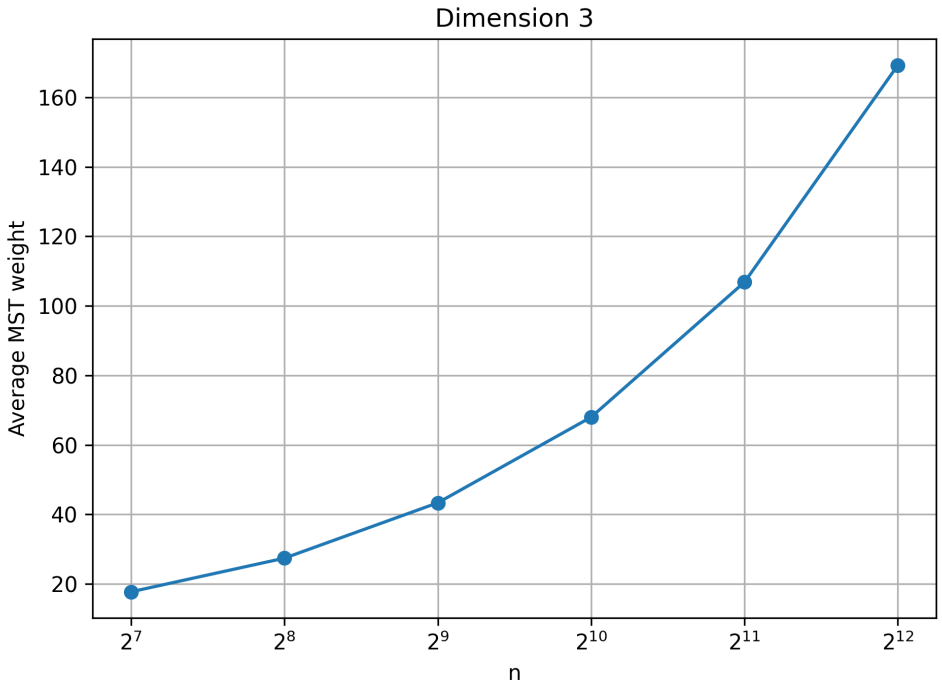

Figure 4: Average MST weight vs. n (Dimension 3).

Fit:
Using $n = 512$ where $512^{2/3} = 64$:

$$c \approx \frac{43.3522}{64} \approx 0.677, \qquad f(n) \approx 0.68 n^{2/3}.$$

Dimension 4

n	trials	average MST weight
128	5	28.3445
256	5	46.8681
512	5	78.0494
1024	5	129.545
2048	5	216.773
4096	5	360.858

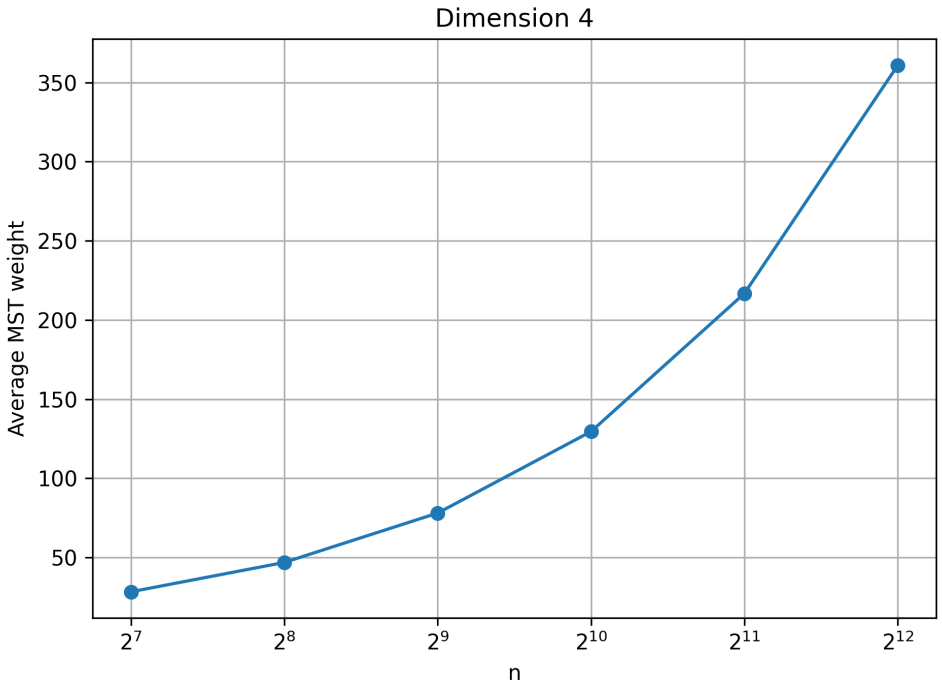

Figure 5: Average MST weight vs. n (Dimension 4).

Fit:
Using $n = 256$ where $256^{3/4} = 64$:

$$c \approx \frac{46.8681}{64} \approx 0.732, \qquad f(n) \approx 0.73n^{3/4}.$$

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.